中国计算机学会教育专业委员会 推荐
全国高等学校计算机教育研究会 出版

高等学校规划教材
国家精品课程教材

Linux操作系统实验教程

罗 宇 陈燕晖 文艳军 等编著

计算机学科教学计划

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

高等学校规划教材

国家精品课程教材

Linux 操作系统实验教程

罗宇　陈燕晖　文艳军　等编著

電子工業出版社

Publishing House of Electronics Industry

北京・BEIJING

内 容 简 介

本书是国防科技大学国家精品课程“操作系统”配套教材，系统地讲解了 Linux 操作系统原理和基于 Linux 的各种编程，特别是 Linux 操作系统内核编程。本书内容分为三部分：第一部分介绍 Linux 操作系统原理；第二部分介绍 10 个基于 Linux 的实验；第三部分附录包含 Linux 环境下的操作及与编程有关的命令和函数列表。

本书适合作为高等学校计算机科学与技术、软件工程、信息安全、信息与计算科学、信息管理与信息系统等专业操作系统实验和课程设计教材，也是 Linux 开发人员熟悉 Linux 环境下应用及内核编程的入门参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Linux 操作系统实验教程 / 罗宇等编著. —北京：电子工业出版社，2009.2
高等学校规划教材
ISBN 978-7-121-08217-7

I. L… Ⅱ.罗… Ⅲ.Linux 操作系统—高等学校—教材 Ⅳ.TP316.89

中国版本图书馆 CIP 数据核字（2009）第 013267 号

策划编辑：童占梅
责任编辑：童占梅
印　　刷：北京市海淀区四季青印刷厂
装　　订：涿州市桃园装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×1 092　1/16　印张：11.25　字数：276　千字
印　　次：2009 年 2 月第 1 次印刷
印　　数：4 050 册　　定价：19.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

操作系统是计算机系统必不可少的关键软件。操作系统教学不但需要讲授操作系统概念、原理与方法，还要让学生动手在实用操作系统上进行编程实践，只有这样才能让学生真正理解操作系统的概念、原理与方法。编写本书的目的就是为了在学习完操作系统原理后，为操作系统实践教学提供指导。

Linux 操作系统是源代码公开的广泛使用的操作系统，利用 Linux 作为操作系统课程的实验平台，不但能帮助学生进行以理解操作系统原理为目的的实验，同时也可以看作是操作系统开发实战的演练。由于 Linux 源代码公开以及 Linux 的广泛普及，我们认为将 Linux 作为操作系统实验教学平台还将为学生毕业后快速进入实战工作状态打下良好的基础。

本书是国防科技大学**国家精品课程“操作系统”建设的成果之一**，之前出版的**《操作系统（第 2 版）》和本书组成了完整的理论和实践教学体系**，并在教学中取得了很好的教学效果。**欢迎登录“操作系统”国家精品课程网站 http:// 618.net.cn。**

本书分为三部分。第一部分介绍 Linux 操作系统原理；第二部分介绍 10 个基于 Linux 的实验；第三部分附录包含 Linux 操作及编程所需的参考资料。本书可作为 CCC2002《操作系统实验和课程设计》专用指导教材，其中部分内容也可以作为操作系统原理课程的课后实验。本书还是 Linux 技术人员的系统编程入门参考书。

本书作者长期从事计算机操作系统研究开发及教学工作，根据多年操作系统开发及国家精品课程建设的教学经验，参考国内外近年出版的各类操作系统实验教程特色，设计了一组基于 Linux 环境的操作系统课程实验，它包括三种类型：系统管理用 shell 脚本编程及用户态运行应用程序实验；内核入门实验；内核综合实验。用户态编程实验主要是让学生体会操作系统功能及接口，在内核实验中，有内核模块实现这样的内核入门实验，也有文件系统、USB 盘驱动这样的实用综合实验，目的是让学生由浅入深地实际体验 Linux 操作系统的系统能力及操作系统设计原理。

本书提供了 Linux 操作系统基本实现原理，实验内容、实验背景知识、解决方案的描述，附录给出了应用和内核编程可能涉及的系统调用、多线程库函数等，读者可以在无须其他参考书的情况下实现基本的实验编程。在操作系统实验课程教学实践中，不需局限于本书所列的实验，建议任课老师将最终实验题（如动态加载模块方式的某驱动程序实现或由学生自选的某个实验）在第一次课时就布置给学生，让学生自己理解实现目标和实验所要掌握哪些基本实验，遇到问题，鼓励学生通过互联网探索解决问题的方法，这样可以发挥学生的主观能动性，以收到更好的实验效果。建议第 8 章实验作为操作系统原理课程的课后实验，第 10 章的实验作为独立操作系统实验课的综合实验由学生任选，而第 9 章实验由学生根据所选综合实验需要自行实验。整个实验教学建议在实验机房进行，**本书所有实验在 Linux 发行版 Fedora 7 及内核版本 2.6.21.7 下经过了验证。**

本书由罗宇负责组织和统稿，罗宇、陈燕晖编写了 Linux 操作系统原理部分，文艳军编写了实验 10，其他部分均由陈燕晖编写。个别实验参考了罗宇、储瑞编著、机械工业出版社出版的《操作系统课程设计》。参加本书编写工作的还有李冬、晏益慧。由于编者水平有限，错误在所难免，有对本书的任何批评和建议可发送邮件到 yuluo@nudt.edu.cn。

编著者

于湖南长沙国防科技大学计算机学院

目　　录

第一部分　Linux 操作系统基本原理

第二部分 基于 Linux 操作系统的实验

第三部分 Linux 环境下的操作及常用命令和函数

第一部分

Linux操作系统基本原理

Part one

第一部分共分 7 章，简要介绍 Linux 操作系统原理。

第 1 章介绍 Linux 的发展历程和开发模式，以及内核代码的组织结构。

第 2 章介绍 Linux 的进程管理机制。Linux 特殊的 clone 系统调用导致它支持线程的方式与操作系统原理介绍的支持线程的方式有着明显的区别，读者需要仔细体会。此外，Linux 调度器采用的算法实际上是多种算法的糅合，请注意与本科教材介绍的算法进行比较。

第 3 章介绍 Linux 的存储器管理。作为支持虚拟存储器的系统，先介绍分页存储采用的伙伴算法，然后介绍内核各子系统广泛用到的 slab 分配器，该分配器的粒度细且效率高。最后介绍进程地址空间。读者应熟悉支撑“分页技术”的核心数据结构，了解相应的硬件支持，而为了提高性能所采用的“写时复制”则是非常实用的技术。

第 4 章介绍 Linux 的文件系统。文件系统为用户提供了抽象易用的接口，如打开、读/写和关闭等，读者应该抓住“这些操作如何映射到具体的物理文件系统”这条主线来贯穿各个知识点。Linux 为了支持多种不同的文件系统，引入了中间层 VFS，VFS 实际上采用了面向对象的设计方法。本章随后介绍了物理文件系统 EXT2 的物理存储布局和核心数据结构，最后讲述几个系统调用的实现。文件系统的性能对系统影响极大，内核设计中对多种对象都采用了缓冲技术，请读者仔细体会。

第 5 章介绍设备管理。由于设备的多样性，本章只介绍一些最基本的概念和核心数据结构。更详细的一些细节则放到了第二部分的实验环节中讲述。

第 6 章介绍中断、异常和系统调用。其中相当一部分知识的介绍依托于 i386 平台，读者在阅读过程中应该注意区分。如果可能，了解另外一种体系结构的相关实现则更有裨益。

第 7 章介绍 Sys V 的进程间通信在 Linux 上的实现。

从整体上来说，Linux 操作系统的实现是操作系统原理教材中所讲述的基本概念、原理及算法的运用，但实际情况是，真正的实现往往比一般原理要复杂得多。内核黑客 Jens Axboe 谈及 I/O 调度算法时说过“The classic IO scheduling algorithms you find in OS text books aren’t really useful in a modern system”，这句话不仅仅适用于 I/O 调度算法。所以学习 Linux 内核原理可看成是进一步的深化学习，如果同时参看源代码，又辅以动手环节，无疑是更加有益的。

第 1 章　Linux操作系统简介

Linux 的发展史极富传奇性。本章首先介绍 Linux 的渊源和发展史，接着介绍 Linux 的特点和开发模式，随后介绍内核代码的组织结构和 Linux 的学习方法。

1.1　Linux的渊源和发展简史

1965 年，美国麻省理工学院、通用电气公司及贝尔实验室联手开发在当时看起来非常先进的 MULTICS 操作系统。经过数年开发后，贝尔实验室认为该项目没有成功的希望便撤出了自己的开发人员，这些人中便有 Ken Thompson 和 Dennis Ritchie。随后，Ken Thompson 在实验室闲置的 PDP-7 机器上开发出了 UNIX 操作系统。最早的 UNIX 是用汇编语言开发的，20 世纪 70 年代初期，Dennis Ritchie 发明了 C 语言并用 C 语言重写了 UNIX 操作系统，直到如今 C 语言依然是系统编程语言的首选。

1974 年，Ritchie 和 Thompson 在 Communications of the ACM 发表了一篇经典论文“The UNIX Time-Sharing System”，使 UNIX 得到广泛关注。1975 年，美国贝尔实验室发布了 UNIX 第六版（V6），当时一些大学及研究机构得到该系统及其源代码，并用于研究和教学。美国加州大学伯克利分校就是依据这个版本开始研究并加以发展的，并在 1977 年发布 1 BSD（1st Berkeley Software Distribution），BSD UNIX 后来成为 UNIX 家族最重要的分支之一。

20 世纪 80 年代初期，UNIX 开始商业化，AT&T 发布了 System Ⅲ和 System V 等。一些商业公司也陆续跟进，发布自己的商用 UNIX 版本，如 IBM 公司的 AIX、Sun 公司的 SunOS（后来命名为 Solaris）等，这些商业版本的 UNIX 源代码不再开放。商业公司的利益角逐导致 1984 年之前 UNIX 蓬勃发展的景象不再复现。

一些有识之士认为，计算机软件的商业化将有碍知识的传播和人类的进步，妨碍用户自由，其中尤以 Richard Stallman 表现最为激烈。Richard Stallman 于 1984 年着手 GNU 计划，计划开发一套与 UNIX 相互兼容且自由开放的软件。1985 年，Richard Stallman 又创立了 FSF（Free Software Foundation）为 GNU 计划提供各方面支持。Richard Stallman 提倡软件自由，为反对软件商业化针锋相对地制订了 GPL（General Purpose License，通用公共许可证）。GPL 力图保证用户共享和修改自由软件的自由，保证自由软件对所有用户是自由的。到 20 世纪 90 年代初，GNU 计划已经发布了数量可观的基于 GPL 的软件，其中包括著名的编译器套件 GCC、编辑器 Emacs 等，这些软件广泛用于各种工作站的商用 UNIX 系统上。但是，GNU 计划并未完全成功，因为它缺少最核心的部分，属于自己的操作系统。

1991 年，芬兰赫尔辛基大学的学生 Linus Torvalds 由于在使用教学型操作系统 Minix 的过程中感到不满意，萌生自己开发一个操作系统的念头。1991 年 9 月，Torvalds 通过网络发布了 0.01 版，后来该系统命名为 Linux 并基于 GPL 发布。Linux 是一个类 UNIX 的操作系统，正好弥补 GNU 计划中缺失的一环，两者相得益彰，所以有 GNU/Linux 的说法。分布在世界各地的许多志愿开发者陆续加入 Linux 内核的开发，使 Linux 的发展速度非常快。1994 年 3 月，Linux 1.0 正式发布。1996 年 6 月，Linux 2.0 发布。1999 年 1 月，Linux 2.2 发布。此时

Linux 已经获得世界性的知名度，以 IBM 为首的许多大型商业公司也大力支持 Linux 的发展，Linux 在操作系统的市场份额也稳步上升。Linux 2.4.0 在 2001 年 1 月发布，Linux 2.6.0 在 2003 年 12 月发布，目前 Linux 内核的最新版本已经到 2.6.28。

Linux 的蓬勃发展也带动了更多的软件以 GPL 许可发布，但是与此同时，自由软件的领袖 Richard Stallman 对商业软件的激进态度导致一些商业公司对此驻足观望。为此 Bruce Perens 和 Eric Raymond 提出了“Open Source”（开放源码）的概念，下面列出 Bruce Perens 定义“Open Source”的 10 条标准：

① 自由再散布。允许获得源代码的人可自由再将此代码散布。

② 源代码。程序的可执行文件在散布时，必须以附带完整源代码或是可让人方便地获得源代码。

③ 衍生著作。对源代码修改后，在依照同一授权条款的情形下再散布。

④ 原创作者程序源代码的完整性。意即修改后的版本，需用不同的版本号码，以与原来的程序代码有所区别，保障原来程序的代码完整性。

⑤ 不得对任何人或团体有差别待遇。

⑥ 对程序在任何领域内的利用不得有差别待遇，意即不得限制商业使用。

⑦ 散布授权条款。若软件再散布，必须用同一条款散布。

⑧ 授权条款不得专属于特定产品。

⑨ 授权条款不得限制其他软件。

⑩ 授权条款必须技术中立。

开放源码相比自由软件的要求相对宽泛一些，如以 GPL，BSD License，LGPL 等许可证下发布的软件均可称为开放源码软件，在一定程度上鼓励了更多的公司和组织加入到开放源码的阵营。

1.2 Linux的基本特性

Linux 之所以能跟各种商业操作系统一争短长，主要在于它丰富的特性。简单概括如下：

① Linux 是免费的，比起 Windows 之类的操作系统，优势一目了然。

② 支持多用户多任务，用户界面良好。多用户多任务是现代操作系统的一个基本特点。早期的 Linux 在图形界面方面跟 Windows 还有一些差距，但是现在差距已经非常微小。而命令行界面向来是 UNIX 系统的强项。

③ 支持多处理。Linux 不仅支持单处理器，而且也支持 SMP（对称多处理，包括近年来的多核芯片）和 NUMA（非一致性存储访问）体系结构。此外，在集群计算领域，Linux 也占据了非常重要的地位。

④ 良好的可移植性。Linux 是支持硬件平台最多的操作系统，这方面远远超过 Windows 等商业操作系统。从掌上设备到微型计算机，从大量的 RISC 工作站再到 IBM 的大型计算机，都可以见到 Linux 的身影。

⑤ 可完全定制，从而灵活应用于各种不同场合。Linux 内核有上千个内核编译选项，用户可根据自身需要选择相应的功能，即可以裁剪成很小的内核用于嵌入式设备，也可以定制成功能丰富的服务器级别内核。用户甚至可以亲自修改内核以满足自己的需要。

⑥ 提供了丰富的网络功能。Linux 本身就是通过 Internet 发展起来，Linux 在通信和网络

功能方面的支持也优于其他操作系统。它不仅支持通用的如以太网、无线网、蓝牙技术等，连不太常见的 DECnet 和 Acorn Econet 等都提供支持。

⑦ 性能出色且安全稳定。Linux 内核开发者总是把性能作为设计的首要目标，许多性能测试比较都证实 Linux 非常高效。Linux 可以连续运转数月乃至数年而无须重新启动，这是许多服务器架设在 Linux 平台上的原因。Linux 采取许多安全技术措施，包括权限控制、审计跟踪、授权机制等。功能强大高效的防火墙 Netfilter/Iptables 能有效防止非法入侵。美国国防部将计算机安全等级划分为四类七级，这七级从低到高依次为 D，C1，C2，C3，B1，B2，B3，A1。Linux 2.6 内核吸收了由美国国家安全局 NSA 开发的访问控制体制 SELinux（Security-Enhanced Linux），使 Linux 的安全级别从原来的 C2 级可以达到 B1 级。

⑧ 良好的兼容性。这表现在多个方面。首先，Linux 是一个与 IEEE POSIX（Portable Operating System Interface）兼容的操作系统。POSIX 标准是为提高 UNIX 系统之间的移植性而制订的。其次，其他操作系统的二进制程序可以直接到 Linux 运行，如 Windows 程序就可以直接跑在 Linux 下的 Wine 模拟器上。此外，许多操作系统的文件分区可以被 Linux 直接访问，如 Windows 的 FAT 分区和 NTFS 分区在 Linux 下均能直接识别，反之 Linux 的 Ext3 分区 Windows 就无法识别。

⑨ Linux 社区支持快速。Linux 社区有良好的运作机制，当用户的问题发到相关新闻组或邮件列表时可能马上得到回应。如果发现 Linux 的 bug 并报告，也许没过多久补丁就出来了。当有新的硬件推出时，Linux 社区的开发人员马上就能提供支持，典型的例子是 Intel 的 Itanium 处理器和 IBM 的 Cell 处理器推出后不久 Linux 就能运行在上面。

1.3 Linux内核的开发模式与内核版本号

内核版本是在 Linus 领导下开发的系统内核的版本号。Linux 内核版本号的递增方式在 2.6.0 前后是有所不同的。下面分别介绍。

在 2.6.0 版本之前，内核版本号以 A.B.C（如 1.2.3，2.3.12，2.4.18，2.5.20）的形式发布，其中：

① A 表示内核主版本。它很少修改，只有代码出现重大变化时才可能变动。目前只变动过两次，1994 的 1.0 和 1996 年的 2.0。

② B 表示内核次版本。如果 B 是偶数，表明该版本是稳定内核，如 1.2.3 和 2.4.18 都是稳定内核。如果 B 是奇数，表明该版本是开发版，不一定可靠，如 2.3.12 和 2.5.20。市场上发行的 Linux 当然都是基于稳定版本的内核，而与开发版内核打交道的主要是内核开发人员。

③ C 表示发布号。以 2.4.18 的发布为例，该版本相对 2.4.17 内核并没有重要特性上的区别。可能只是修订一些 bug 和安全上的漏洞，还可能添加了一些硬件驱动。但是开发版本不是这样，见下面的说明。

重大特性的修改在稳定版本中基本是不会出现的，因为可能导致内核不稳定。如果添加重要特性，必须在开发版本中进行，我们以 2.4 到 2.6 的演进过程为例说明，2.4.0 发布后，Linus 持续发布 2.4.x 稳定版本，一直到 2.4.15。Linus 把 2.4.15 后续版本的维护工作交给 Marcelo Tosatti，而 Linus 基于 2.4.15 分支开发版本 2.5.0。开发版本允许开发者对内核尝试全新的特性，如果被证实有效，则可能被吸收进 2.5.x，像 O（1）调度和可剥夺内核特性都是这样加入的，当然加入的新特性如果发现有不足之处也可能在后面的版本中被剔除。众多内核开发

者的主要注意力都集中在 2.5.x，使得 2.5.x 的新功能快速增加，发布号的变迁也非常快。当 Linus 发布 2.5 系列的最后一个开发版本 2.5.75 后，基于该版本 Linus 终于发布了全新的（相对 2.4.x 而言）稳定内核版本 2.6.0。

上述开发方式的一个缺点是发布周期太长，从 2.4 到 2.6 花了将近三年时间，企盼新特性的 Linux 发行商和广大用户都不愿意见到这种情况，于是 2.6.x 内核的开发模式就发生了变化。

在 2.6.0 版本之后，所有稳定内核版本号以 2.6.C[.D]（如 2.6.8，2.6.8.1）的形式发布。C 编号的变化可能意味着内核代码的大量修改以及重大特性的引入。数字 D 的引入主要是处理严重的错误或修订 bug 以及安全补丁，如 2.6.8.1 的发布是修改 2.6.8 版本中 NFS 代码的错误。

在 2.6 开发过程中，Andrew Morton 的地位仅次于 Linus Torvalds，他维护 2.6.x-mm 分支，该分支包含了大量补丁，如果一个补丁在-mm 分支里证实了自己的价值，Andrew 或者子系统维护者就会把它提交给 Linus，以求被收录于 Linus 发布的内核。为叙述方便，我们以 2.6.26 到 2.6.27 的发布说明其开发流程。在 2.6.26 发布后，2.6.26.x 的发布由一个稳定版小组而非 Linus 本人负责，与此同时，大约两周的时间合并窗口被打开，数量众多的补丁（可以包含新特性而且大部分在-mm 树中经过了一段时间的测试）被提交给 Linus，然后是 2.6.27-rc1 的发布（rc 是 release candidate 的缩写）。在 rc1 发布后，涉及新特性的补丁暂时不被接纳，只能等到发布 2.6.28-rc1 的时候。但是其他的补丁依然可以进入 2.6.27-rcx（x 为 2，3，…）系列。Linus 一般每周发布一次 2.6.27-rcx，经过多次这样的发布，当 rc 系列逐渐稳定后，如 2.6.27-rc9 发布后，2.6.27 正式发布，同时 2.6.28-rc1 的合并窗口开启。这种开发模式每年可以发布多个 2.6.x 版本，极大地缩短了发布周期。

1.4 Linux发行版介绍

人们常说的 Linux 可能有两个含义，一个指 Linus 领导发布的内核，另一个指 Linux 的发行版。对于普通用户来讲，即使有内核依然做不了什么事情，还需要各式各样的软件。目前有大量软件是开源的，下面列举若干：

- Web 服务器，如 Apache。
- 数据库，如 MySQL，PostgreSQL。
- 网页设计语言，如 PHP。
- 图像处理软件，如 GIMP。
- 网页浏览器，如 Firefox。
- 办公套件，如 OpenOffice。
- 邮件，如 Thunderbird。
- 开发工具与库：GCC，GDB，Glibc。

Linux 发行版可能是由一个组织、公司或个人发行的。通常来讲，一个典型的 Linux 发行版包括：Linux 内核，GNU 工具和库，图形界面的 X Window 系统和相应的桌面环境，如 KDE 或 GNOME，还包括其他的自由软件或开源软件。一些商业 Linux 发行版中也有一些专有软件。发行版往往基于不同的目的而制作， 例如，国内本土化的红旗 Linux、应用嵌入式场合的 uclinux 或着重桌面用户体验的 Fedora 等。目前 Linux 的发行版已经有几百种，下面介绍几个比较流行的发行版。

① Debian。最早由 Ian Murdock 于 1993 年发布，现在由来自世界各地的志愿者开发和

维护，它不属于任何的商业公司，完全由开源社区所有。Debian 软件包管理工具 dpkg 非常强大，在 Debian 上安装、升级、删除和管理软件极为容易。Debian 以稳定性闻名，所以很多服务器都使用 Debian 作为其操作系统。追求稳定性导致 Debian 新版本的发布周期较长，较少桌面玩家会选用它。

② Fedora。是由红帽公司赞助，由开源社区与红帽公司合作开发的项目。Fedora 的定位主要是桌面用户，对于使用者而言，它是一套功能完备、更新快速的免费操作系统，而红帽公司则将它作为许多新技术的测试平台，被认为可用的技术最终会加入到它的商业发行版 Red Hat Enterprise Linux 中。Fedora 大约每 6 个月发布一个新版本，目前最新版本是 Fedora 10。

③ Ubuntu。由 Mark Shuttleworth 在 2004 年 10 月发布首个版本，目前最新版本是 8.10。Ubuntu 基于 Debian 发行版和 GNOME 桌面环境，与 Debian 的不同在于它每 6 个月会发布一个新版本。Ubuntu 十分重视系统安全，所有系统相关的任务均需使用 sudo 指令，这是它的一大特色，这种方式比传统的以系统管理员账号进行管理的方式更安全；Ubuntu 还十分注重系统的易用性，标准安装完成后，一般就可以投入使用。

④ Suse。原是以 Slackware Linux 为基础，并提供完整德文使用界面的产品，在欧洲市场占有相当份额。2004 年 1 月被 Novell 收购，目前最新版为 Suse Linux 11.0。Suse 在安装时能自动调整 NTFS 分区，为初学者减少了不必要的麻烦。

1.5 Linux内核源代码组织结构

Linux 内核源代码可以从官方站点 http://www.kernel.org/下载。下载的源代码是压缩包格式，可以在 Linux 或者 Windows 环境下解压到某个目录，可以看到该目录下包含许多子目录，下面对它们进行简单介绍。

Arch——包含体系结构相关的代码，每种体系结构都有一个相应的子目录。如 x86 相关的代码放在 i386 目录下。

Block——Block I/O 层的代码，包含多种磁盘 I/O 调度算法。

Crypto——各种加密算法。

Documentation——与内核相关的文档，组织比较零乱，涉及面很广。但有些非常有用，介绍了内核某些功能模块的设计原理。

Drivers——各种设备驱动程序。

Fs——内核支持的各种文件系统，如 EXT3，NTFS 等。

Include——包含了绝大部分内核头文件。

Init——内核启动和初始化代码。

Ipc——进程间通信代码。

Kernel——最核心部分，包括进程管理、同步原语的实现等。

Lib——内核的辅助函数。

Mm——存储管理子系统，与平台相关的部分在 arch/*/mm 目录下。

Net——网络子系统，包含多种网络协议的实现。

Scripts——包含构建内核的脚本文件。

Security——包含 SELinux 的实现。

Sound——音频子系统。

Usr——EarlyUserSpace 特性的相关代码。

1.6 学习Linux的辅助软件介绍

使用本书的前提当然是读者手头有可用的 Linux 发行版，本书绝大部分内容适用于各种发行版，但极小一部分内容是基于 Fedora 的，所以**推荐使用 Fedora**。现在的 Linux 发行版安装已经非常简单，即便是新手，按照安装向导的提示一路下去也很容易成功。对很多初学者安装最大的困扰是一开始机器安装了 Windows，没有为 Linux 预留磁盘分区，解决该问题的办法有很多，一种简单的方法是在 Windows 里面安装虚拟机软件，再在该虚拟机中安装 Linux。

虚拟机产品有很多，如 VMware，Virtual PC 和 Bochs 等。笔者试用的是 VMware 工作站版本，可在网址 http://www.vmware.com/download/desktop_virtualization.html 下载。这个软件可在 Windows 环境下模拟出一台甚至多台新的计算机，然后便可在这些新计算机上面安装任何操作系统，包括 Windows 和 Linux，而且对目前的操作系统没有影响。如图 1-1 所示，笔者的虚拟机是 VMware Workstation 6.0，虚拟机里面安装了两个 Linux 发行版，Redhat Linux 8.0 和 Fedora Core 5。使用虚拟机对内核编程有特别的好处，当编程错误导致内核崩溃时，机器重启只是软件重启而不是硬件级别的重启，用户依然可以做其他的工作。其次使用虚拟机的快照（snapshot）功能，系统可以快速启动。VMware 的具体安装和使用都比较简单，请读者自行参考有关文档，这里就不详细介绍了。

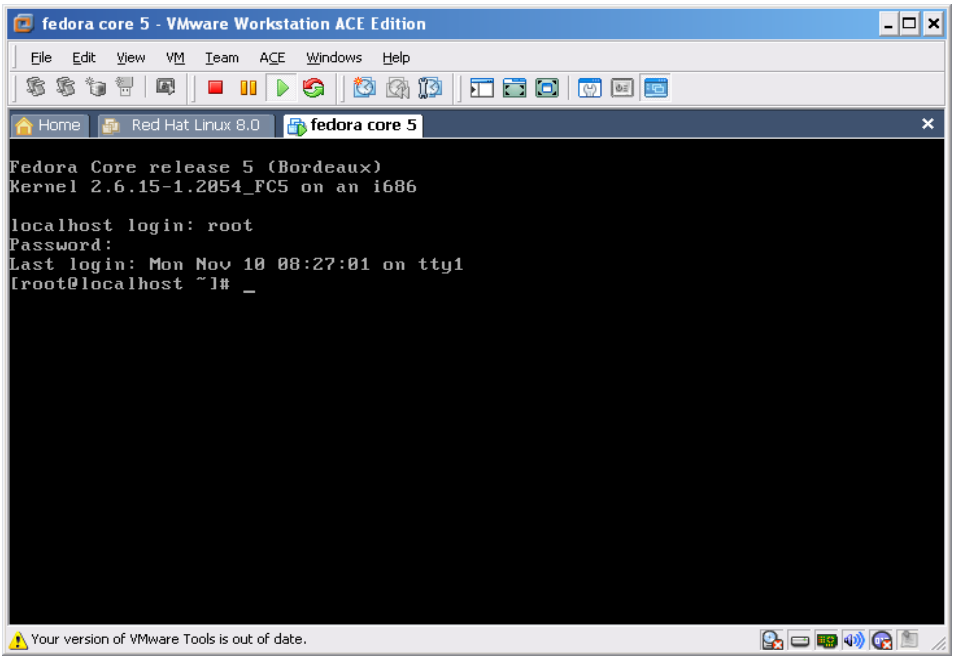

图 1-1　虚拟机 VMware Workstation

在此要介绍的第二个工具软件是 Source Insight，它在阅读 Linux 源代码的时候特别有用。Source Insight 可以从 http://www.sourceinsight.com/downeval.html 下载，运行该软件可免费试

用 30 天。Source Insight 的安装和使用都非常简单，支持 C/C++，C#，Java 等多种高级语言。安装之后，选择功能菜单上的【Project】选项的子菜单【New Project】新建一个项目，把欲读的源代码加入，这个软件会花一点时间分析用户所加的源代码。分析完之后，就可以阅读了。如果在阅读中遇到哪个全局变量、宏或者函数，只要把光标定位于该变量或函数，其定义就会马上在一个子窗口中显示出来，见图 1-2。这个软件还支持语法制导的颜色显示，源代码结构浏览等很多实用的功能。

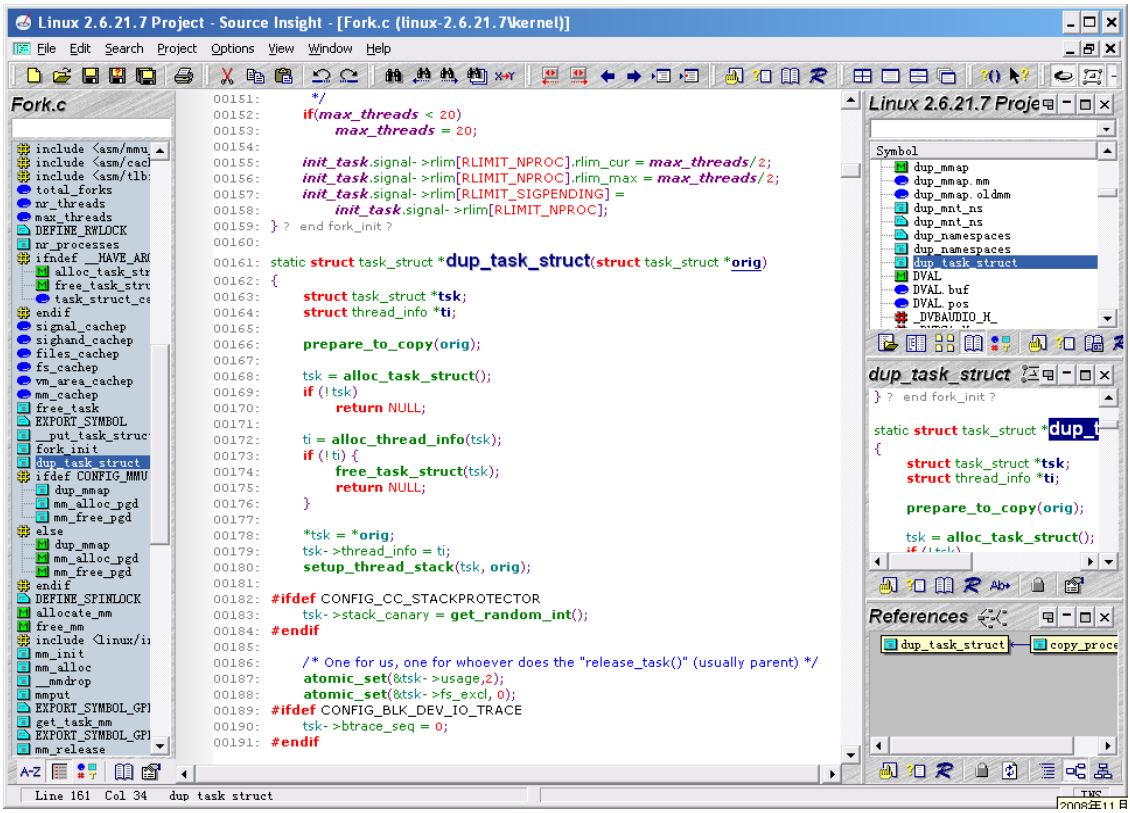

图 1-2　Source Insight 界面

Source Insight 只能在 Windows 环境下使用。另外一种浏览源代码的方式是访问站点 http://lxr.linux.no/，该站点可以浏览多个版本的源代码，强大的交叉索引功能可以帮助用户快速找到相关的标识符。

第 2 章　Linux的进程管理

Linux 支持传统的进程概念，也支持线程的实现，但和传统的做法有所区别。本章先介绍进程描述符，然后介绍进程的切换时机及调度算法，最后描述进程的创建与消亡过程。

2.1　进程与进程描述符

支持线程的传统做法是把进程看成内核资源分配的单位，而线程是调度的基本单位，内核有专门的数据结构分别描述进程与线程。Linux 的做法比较特别，Linux 支持传统 UNIX 进程的概念，进程控制块在 Linux 中称为进程描述符（Process Descriptor），由 task_struct 结构表示。每个进程都有一个进程描述符，进程不但拥有资源而且参与调度。Linux 内核没有明确的数据结构表示线程，但 Linux 通过 clone( )系统调用支持轻量级进程（Lightweight Process）的概念，子进程可以和父进程共享地址空间、打开文件表等信息，轻量级进程也使用 task_struct 结构描述，同样参与调度。Linux 2.6 平台主流线程库 NPTL 的实现就借助了 clone ()系统调用。

Linux 还支持内核线程的概念，内核线程永远在核心态运行，共享内核地址空间，没有用户地址空间。页面换出、刷新磁盘缓冲等工作都由内核线程完成。

task_struct 是一个复杂的结构，占一千多字节，其各个成员用来准确描述进程在各方面的信息，主要有以下 8 个部分：

（1）进程标识。包括进程的标识号（pid）、进程的用户标识、进程的组标识等。每个进程的标识号是唯一的。

（2）运行环境信息。指向结构 thread_info 的指针，该结构包含当前进程运行的一些环境信息，其中有指向所在进程描述符的指针。

（3）调度相关信息。这部分内容与进程调度有关，一部分信息见 2.2 和 2.3 节。进程描述符中还需要有结构地保存当前进程被换出时寄存器的状态，该进程恢复运行时便可从原有的状态继续运行。

（4）进程地址空间信息。Linux 的进程都在自己的私有地址空间中运行，task_struct 的成员 mm 指向一个 mm_struct 结构，见第 3 章。

（5）文件相关信息。包含进程与文件系统交互的信息，见 4.1 节。

（6）信号处理信息。Linux 支持传统的 UNIX 信号语义。该部分记录了信号的处理函数及信号掩码等信息。

（7）记账信息及统计信息。资源是有限的，每个进程对每种资源的使用都有一个限值。另外，还有统计信息来记录系统需要的信息，如页面异常次数、CPU 使用时间等。

（8）描述进程间关系的指针。进程并不是孤立存在的，所有的进程通过一个双向链表链接在一起。通过宏 for_each_process 就可以对进程进行遍历。进程描述符还拥有指向其父进程描述符、子进程描述符、兄弟进程描述符的指针。此外，很多场合需要根据 pid 号能够快速找到进程，系统以 pid 为关键字建立了一个哈希表，哈希函数值相同的进程通过进程描述符的 pids 成员链在一起。

Linux 的运行分为两种模式——核心态和用户态。内核总在核心态下运行，而进程通常在用户态运行，只有通过系统调用或者中断异常被触发时才能切换到核心态运行。

进程拥有两个栈——用户模式栈与核心模式栈，分别在相应模式下使用。进程描述符的运行环境信息 thread_info 和进程核心栈的空间分配在一起，内核编译选项 CONFIG_4K_STACK 决定它们占用一个还是两个连续的物理页帧，而 task_struct 对象则利用 slab 分配器进行分配。

图 2-1 可以清楚说明这一点。

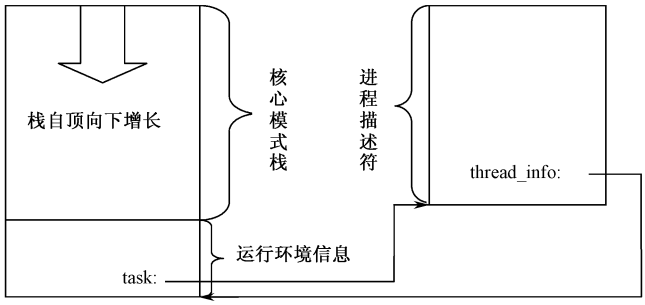

图 2-1　核心栈、进程描述符与运行环境信息

2.2 进程状态及切换时机

2.2.1 Linux的进程状态

Linux 的进程状态有 7 种，由变量 state 和 exit_state 表示。state 表示运行状态，包括 TASK_RUNNING，TASK_INTERRUPTIBLE，TASK_UNINTERRUPTIBLE，TASK_STOPPED 和 TASK_TRACED；exit_state 表示进程退出状态，包括 EXIT_ZOMBIE 和 EXIT_DEAD。

① TASK_RUNNING——表示进程具备运行的资格，要么正在运行，要么就是等待被调度执行。进程描述符有一个 run_list 成员，所有处于 TASK_RUNNING 状态的进程都通过该成员链在一起，称为可运行队列。

② TASK_INTERRUPTIBLE 和③TASK_UNINTERRUPTIBLE——这两种状态均表示进程处于阻塞状态。进程进入阻塞状态一般是因为所请求的资源不能满足。不同的是，TASK_INTERRUPTIBLE 除了资源满足时可以被唤醒外，还可以被信号唤醒，而 TASK_UNINTERRUPTIBLE 就不行。

④ TASK_STOPPED——进程处于暂停状态，当进程收到 SIGSTOP，SIGTSTP，SIGTTIN 或 SIGTTOU 信号时进入该状态。

⑤ TASK_TRACED——被调试进程收到信号时进入该状态并通知调试器，等待调试器发出继续的指令。

⑥ EXIT_ZOMBIE——表示进程已经结束运行，但是父进程还未触发 wait4( ) 或 waitpid( ) 系统调用。

⑦ EXIT_DEAD——进程的终态。进入该状态有两种情况，一是该进程不需要等待父进程来回收进程描述符和核心栈；二是父进程正在对其进程 wait4( ) 或 waitpid( )系统调用。

进程状态转换如图 2-2 所示。

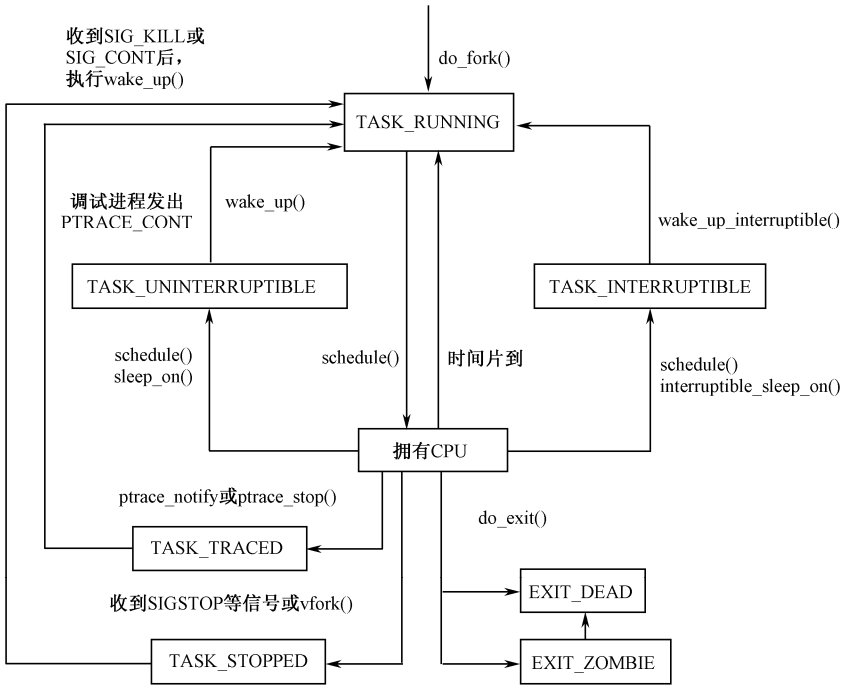

图 2-2　Linux 进程状态转换图

2.2.2　进程的切换时机

当前进程放弃 CPU 从而使其他进程得到运行机会的情况分为两种：进程主动放弃 CPU 和被动放弃 CPU。

进程主动放弃 CPU 又分为两类。

第一类是隐式地主动放弃 CPU。这往往是因为需要的资源目前不能获取，如出现在执行 read()，select()等系统调用的过程中。这种情况下的处理过程如下：

（1）将进程加入合适的等待队列。

（2）把当前进程的状态改为 TASK_INTERRUTIBLE 或 TASK_UNINTERRUTIBLE。

（3）调用 schedule()函数，该函数的执行结果是当前进程放弃 CPU。

（4）某个事件使得进程进入 TASK_RUNNING 状态，经过调度重新获得 CPU，检查资源是否可用，如果不可用，则跳转到第(2)步。

（5）资源已可用，将该进程从等待队列中移去。

第二类是显式地主动放弃 CPU，如系统调用 sched_yield()，sched_setscheduler()，pause()及 nanosleep()均会导致当前进程让出 CPU。

进程被动放弃 CPU 又分成两种情形，其一是当前进程的时间片已经用完，其二是刚被唤醒进程的优先级别高于当前进程。两种情形均会导致当前进程描述符中进程信息的 flags 的 TIF_NEED_RESCHED 标志位置位。

从进程调度时机的角度来讲，当前进程放弃 CPU 也有两种情形。一种是直接调用

schedule()调度函数，如上面提到的进程主动放弃 CPU 的第一类情形；另一种是间接调用 schedule()调度函数，如进程被动放弃 CPU 的情形。当进程描述符运行环境信息中的 flags 的 TIF_NEED_RESCHED 标志位置位时，并不立即直接调用 schedule()调度函数，而是在随后的某个时刻，当进程从内核态返回用户态之前检查 TIF_NEED_RESCHED 是否置位，如果置位，则调用 schedule()调度函数。

内核编译选项 CONFIG_PREEMPT开启时Linux 2.6 内核是可剥夺（Preemptive）的，可剥夺内核在更多的内核代码点检测IF_NEED_RESCHED是否置位，从而让刚被唤醒的高优先级进程尽快得到CPU，减少了分派延迟（Dispatch Latency）。

2.3 进程的调度算法

Linux 进程调度的核心函数是 schedule()，该函数的主要任务是选出一个可运行的进程并将处理机切换到该进程。调度算法属于动态优先级算法，优先级在 0～MAX_PRIO-1 之间取值（MAX_PRIO 定义为 140）。其中 0～MAX_RT_PRIO-1（MAX_RT_PRIO 定义为 100）属于实时进程范围，MAX_RT_PRIO～MAX_PRIO-1 属于非实时进程。数值越大，优先级越小。调度器为每一级的优先级都准备一个可运行队列。随着进程的优先级的改变，进程也将挂入到对应不同优先级的可运行队列。

进程描述符主要有如下成员与调度有关：

（1）policy　进程的调度策略，它有以下三种类型：

① SCHED_NORMAL——非实时进程。

② SCHED_FIFO——实时进程，采用先进先出的调度算法。

③ SCHED_RR——实时进程，采用轮转法。

（2）rt_priority　实时进程的优先级。

（3）static_prio　非实时进程的静态优先级，决定进程初始时间片大小。

（4）sleep_avg　进程的平均等待时间，决定非实时进程动态优先级的主要因素。

（5）prio　进程的动态优先级。

动态优先级的计算主要在 effect_prio() 函数中完成，该函数的实现相当简单。下面是effect_prio()的实现代码：

```
static int effective_prio(task_t *p)
{
    int bonus, prio;
    if (rt_task(p))
        return p->prio;

    bonus = (NS_TO_JIFFIES(p->sleep_avg) * MAX_BONUS /MAX_SLEEP_AVG)
        - MAX_BONUS / 2;
    prio = p->static_prio - bonus;
    if (prio < MAX_RT_PRIO)
        prio = MAX_RT_PRIO;
    if (prio > MAX_PRIO-1)
        prio = MAX_PRIO-1;
```

```
    return prio;
}
```

显然，非实时进程的优先级仅决定于静态优先级 static_prio 和平均等待时间 sleep_avg 两个因素，而实时进程的动态优先级不通过计算改变，但可以通过 sched_setscheduler()设置。

动态优先级不是统一在调度器中计算和比较，而是独立计算。只要进程的状态发生了改变，内核就可能重新计算和设置进程的动态优先级。具体时机包括进程创建、进程被唤醒、时间片耗尽和其他原因。

2.4 进程的创建与消亡

pid 为 0 的 idle 进程是 Linux 系统启动过程中产生的第一个进程。idle 进程会创建一个内核线程，该线程进行一系列初始化动作后最终会执行/sbin/init 文件，执行该文件的结果是运行模式从核心态切换到了用户态，该线程演变成了用户进程 init，pid 为 1。init 进程是一个非常重要的进程，一切用户态进程都是它的后代进程。

UNIX 系统执行新任务的典型方法是通过 fork()/exec()函数。通常 fork()创建一个新进程，然后新进程通过调用 exec 系列函数执行真正的任务。下面是一段示例代码：

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main(void)
{
    pid_t    pid;
    if ((pid = fork()) < 0){
        printf("fork failed\n");
        exit(1);
    }
    else if (pid = = 0){ /*子进程执行进入此部分*/
        execlp("echoall","echoall",(char*)0);
    }
    else{    /*父进程*/
        printf("fork success\n");
        exit(0);
    }
}
```

如果函数 fork()调用成功，当前进程就拥有了一个子进程。该函数返回两个值，其中在子进程中返回 0，在父进程中返回的是子进程的 pid 值。

用于实现创建子进程的系统调用有三个：clone()，vfork() 和 fork()。这三个系统调用最终都会调用 do_fork()函数完成主要工作。该函数的第一个参数 clone_flags 由多个标志位组成，常见的标志位有：

CLONE_VM——子进程和父进程共享进程地址空间。

CLONE_FS——子进程和父进程共享文件系统信息。

CLONE_FILES——子进程父进程共享打开的文件。

CLONE_VFORK——当 vfork()函数被调用时该标志位被设置。

clone()对应的 clone_flags 值可能是多个标志位的组合，这取决于具体情况。vfork()对应的 clone_flags 值是 CLONE_VFORK|CLONE_VM|SIGCHILD，fork()对应的 clone_flags 值是 SIGCHILD。SIGCHILD 的作用是子进程终结或暂停时给父进程发信号。

vfork()是一个老的函数调用，子进程共享父进程的地址空间包括页表，父进程被挂起直到子进程执行 exec 系列函数或子进程退出时。在合适的场合，与 fork()时的“写时复制” 策略（见 3.2 节）相比，vfork()的开销更小。

do_fork()执行的过程大致如下：

（1）调用 alloc_pidmap()为子进程分配一个 pid。

（2）调用 copy_process()，该函数完成子进程创建的大部分工作。

① 调用 dup_task_struct()，创建一个新的 task_struct 结构。dup_task_struct()调用 alloc_task_struct()为子进程描述符分配空间，调用 alloc_thread_info()为子进程分配核心栈，核心栈中包含了 thread_info 结构，组织好相关的指针，并复制父进程的进程描述符和运行环境信息。

② 检查是否超过了资源限制，如果是，结束并返回出错信息，否则更改一些统计量的信息。

③ 调用 audit_alloc()分配进程的记账信息，并初始化。

④ 依次调用 copy_semundo()，copy_files()，copy_fs()，copy_sighand()，copy_signal()，copy_mm()，copy_namespace()来分别复制父进程的 semundo（进程退出时应该执行操作的一个信号量队列）、文件处理、信号处理、进程地址空间信息和名字空间。以上函数的具体行为取决于 clone_flags 参数。例如调用 copy_mm()时，如果 clone_flags 包含 CLONE_VM 标志，则子进程只共享父进程的空间，不会进行复制。

⑤ 调用 copy_thread()初始化子进程的核心模式栈，核心栈保存了进程返回用户空间的上下文。此处与平台相关，以 i386 为例，其中很重要的一点是存储寄存器 eax 值的位置被置 0，这个值就是执行系统调用后子进程的返回值。

⑥ 调用 sched_fork()设置进程的调度信息。sched_fork()将进程的状态将置为 TASK_RUNNING，并将父进程的时间片余额分一半给子进程。

⑦ 调用 set_task_cpu 设置进程所在的处理器编号。子进程和父进程在同一处理器上。

⑧ 利用宏 SET_LINKS 将子进程插入所有进程都在其中的双向链表。调用 attach_pid()将子进程加入相应的 Hash 队列。

（3）如果 clone_flags 包含 CLONE_STOPPED 标志，将进程的状态改为 TASK_STOPPED，否则调用 wake_up_new_task()函数，wake_up_new_task()最终调用__activate_task()将子进程加入可运行队列。至此，子进程创建完毕并在可运行队列中等待被调度运行。

（4）如果 clone_flags 包含 CLONE_VFORK 标志，则将父进程挂起直到子进程释放进程地址空间。

（5）返回子进程的 pid 值，该值就是系统调用后父进程的返回值。

进程运行结束后最普遍的被销毁情况是，进程正常运行结束后显式或隐式调用 exit()函数。当进程收到某种信号，该信号的处理函数会结束进程运行并将其销毁，对于最常使用的

shell 命令之一 kill，在不指定信号的情况下，默认信号 SIGTERM 将被发给进程导致进程结束运行。另一种常见的情况是，用户程序访问了非法空间，使内核向进程发送 SIGSEGV 信号从而导致进程结束运行。进程销毁的最终动作是调用函数 do_exit()来完成的。

do_exit()执行的流程如下：

① 检测进程当前是否处于中断处理中，如果是，给出出错信息并中止系统。

② 检查进程 pid 是否为 0 或 1，即进程是否为 idle 进程或 init 进程，如果是，则给出出错信息并中止系统。

③ 设置标志表明进程正在被销毁。

④ 如果进程在定时器队列中等待，则将其移出。

⑤ 调用 exit_mm()，exit_sem()，__exit_files()，__exit_fs()，exit_namespace()和 exit_thread()释放进程所占用的各种资源。被释放的资源一般先将其共享计数器减一，如果此时还有别的进程使用该资源，则共享计数器不为 0，此时直接返回。如果为 0，才真正释放资源。exit_mm()的动作是释放进程地址空间，释放过程中会检测该进程是否由 vfork()创建，如果是，则唤醒父进程。

⑥ 设置进程的退出码，调用 exit_notify()处理该进程与其父进程和子进程的各种关系。在该函数中，会将该进程的退出状态 exit_state 置为 EXIT_ZOMBIE 或 EXIT_DEAD。

⑦ 调用 schedule()调度函数切换到别的进程。

因为父进程往往要通过 wait()之类的系统调用来检查子进程是否结束，即使子进程结束了，子进程描述符和其关联的部分数据结构也不能立即释放，此时子进程处于 EXIT_ZOMBIE 状态，称之为僵尸子进程。等到父进程执行 wait()系统调用后，僵尸子进程占用的资源才会彻底回收。如果父进程先于子进程结束，这些子进程包括正在运行的子进程的僵尸子进程（但是父进程没有对其进行 wait），这时 init 进程就变成了这些子进程的父进程，init 进程回收它的僵尸子进程所占用的资源。

有些情况父进程不关心子进程的退出信息，这种情况下子进程结束时设置为 EXIT_DEAD 状态，release_task()函数立即被调用。

第 3 章　Linux的存储器管理

Linux 的存储器管理由两部分组成。第一部分是物理内存的管理，第二部分是虚拟存储器的管理，主要是进程虚拟地址空间的管理。

3.1　物理内存的管理

存储管理子系统要管理所有的物理内存，在系统启动时一部分物理内存固定用来存储内核映像和初始化数据；其余的物理内存都是动态分配的，用来满足内核各子系统的内存需求和用户进程的需求。存储分配器必须能够快速响应客户请求，而且要求尽可能地在提高内存利用率的同时减少内存碎片问题。

Linux 核心内存管理采用了基于区域的伙伴系统及 slab 分配器。

3.1.1　页帧与区域

物理内存是以页帧(Page Frame)为基本单位，页帧的大小固定，对 i386 默认为 4KB。每个页帧由一个 struct page 结构描述。2.6 内核支持 NUMA 结构，NUMA 结构的物理内存逻辑上统一编址，但却可能物理上不位于一处，从而导致访问不同位置所需的时间并不一样。访问时间相同的物理区域称为一个结点。为了保证效率，一般不希望有跨结点操作，并且尽可能使用访问时间短的结点。对应 x86，实际只有一个结点。

每个结点的物理内存因为用途不同又分成不同的区域（zone）。例如 i386，分成如下三个区域：

（1）ZONE_DMA。这部分内存是低于 16MB 的内存，是以 DMA 方式能够访问的物理内存。在内存分配时，尽可能保留这部分内存以供 DMA 方式使用。

（2）ZONE_NORMAL。这部分内存是介于 16MB～896MB，直接被内核映射。

（3）ZONE_HIGHMEM。这部分是高端内存，为超过 896MB 以上的部分，不能被内核直接映射。

在内存分配时，首先要选定结点，然后根据区域访问优先级别来决定访问次序。比如为 DMA 方式分配内存，ZONE_DMA 是唯一符合要求的区域。如果用户进程申请页面，则可先到高端内存区分配，如果不能满足要求，再尝试 ZONE_NORMAL，如果还不能满足要求，则求助于 ZONE_DMA。

3.1.2　伙伴算法

内存管理系统有所谓的“外部碎片”问题，即频繁的分配与回收物理页面会导致大量的连续且小的页面块夹杂在已分配的页面中间，当需要一大块连续的物理页面时，即便空闲物理内存总数很多却依然满足不了要求。在每个区域，Linux 对空闲内存的管理采用伙伴算法。

我们以 16 个页面为例，描述它的分配回收过程。假设有 16 个页面，编号从 0 到 15，则可以用 5 个链表来描述页面的空闲情况，分别描述 1，2，4，8，16 个连续空闲页面的情况。

若一开始均为空闲，则只有第 5 个链表不为空，表示页面[0-15]均空闲。

现在申请 2 个页面，先在第 2 个链表中找，发现为空，再到第 3 个链表中，仍为空，再向上找，直到第 5 个链表，发现有 16 个连续空闲页面。清空链表 5 的对应项并把 16 个页面剖为两半，[8-15]加入链表 4，[0-7]继续剖为两半，[4-7]加入链表 3，[0-3]继续剖为两半，[2-3]加入链表 2，页面[0-1]用来满足申请。假设继续申请 4 个页面，先到链表 3 查找页面[4-7]，则将这 4 个页面分配出去，同时清空链表 3 的对应项。

如果用户接着释放页面[0-1]，系统并不将其插入链表 2，而是看它的伙伴[2，3]是否空闲，如果是，则表明可组成一个更大的连续页面[0-3]。所以此时将[2，3]从链表 2 中摘除。继续查看空闲页面[0-3]的伙伴[4-7]是否空闲，[4-7]已被分配，表明不可能合成更大的连续空闲页面，此时才将页面[0-3]加入链表 3。

如果按照上面的例子，两组连续页面块被认为是一对“伙伴”必须满足如下条件:

（1）大小相同，比如说都有 b 个页面。

（2）物理空间上连续。

（3）位置居前的页面块首页编号必须是 $2b$ 的倍数。

页面[4-7]、[8-11]因为不满足条件（3），因而不是伙伴。

从上面分析可以看到，分配页面时尽量动用小的连续页面，回收页面时则尽可能将空闲的伙伴合成大的连续页面，从而很大程度解决了内存的“外部碎片”问题。

Linux 的每个区域的空闲内存实际是挂在 MAX_ORDER(定义为 11)个链表中（大小从 2^0-2^{10} 页帧）。直接向伙伴系统申请空间的函数为__alloc_pages()，而对应的释放函数为__free_pages_ok()。当在申请内存发现页面短缺时，系统还会唤醒 kswapd 内核线程运行，该线程会腾出一些空间以满足要求。

3.1.3 slab分配器

伙伴系统是以页帧为基本分配单位，在很多情况下不符合实际需要。比如文件子系统要使用的 inode 对象、文件对象都远远小于页的大小，这种情况下分配一个页面无疑会造成很大的浪费，这就是内部碎片的概念。Linux 引入了 SunOS 操作系统中的技术——slab 分配器。slab 建立在伙伴系统之上，能更好地利用内存来提高性能。

slab 分配器的基本思想是，为经常使用的对象建立缓冲池，申请一个对象时，如果缓冲池有空闲对象就分配一个给申请者，缓冲池无空闲对象时才向伙伴系统申请更多的对象。释放对象时将该对象加入缓冲池即可。这样做的好处是，申请与释放操作主要与缓冲池交互，尽可能少地与伙伴系统打交道，从而提高了效率。

slab 分配器为不同的常用对象生成不同的缓冲池，每个缓冲池存储相同类型的对象。但对象缓冲池并非由各个对象直接构成，而是由一连串的 slab 分配器构成，每个 slab 分配器又由一个或多个连续的物理页帧组成，包含了若干同种类型的对象。假定 slab 分配器大小为 slab_size，对象大小 obj_size，一个 slab 分配器可以容纳 N 个对象，产生的碎片为 frag(frag<obj_size)，则有如下公式：

$$\text{slab\_size} = N \times \text{obj\_size} + \text{frag}$$

可以得出内部碎片率 frag/slab_size 的值不超过 $1/(N+1)$，Linux 的实现保证碎片率不超过 12.5%，该值在实际系统中是可以接受的。

对象缓冲池的控制结构为 kmem_cache_t。因为各种对象缓冲池都有一个 kmem_cache_t

结构管理，系统专门建立了变量名为 cache_cache 的缓冲池，该缓冲池存放类型为 kmem_cache_t 的对象。

除了上面讨论的特定对象的缓冲池外，Linux 还提供了 13 种通用的对象缓冲池，其存储对象的大小分别为 32B，64B，128B，256B，512B，1KB，2KB，4KB，8KB，16KB，32KB，64KB 和 128KB。这些缓冲池用来满足特定对象之外的普通内存需求。其单位的大小呈级数增长保证了内部碎片率不超过 50%。

下面介绍 slab 分配器的相关操作。

（1）kmem_cache_create()函数

该函数从 cache_cache 缓冲池中得到一个空闲 kmem_cache_t 对象，然后初始化各成员项。这需要进行一系列运算，以确定最佳的 slab 分配器构成。包括：每个 slab 分配器由几个页面组成，可包含几个对象，slab 分配器的控制结构应该在 slab 分配器的内部还是外部存放。下面的代码创建了名为 inode_cache 的对象缓冲池，该缓冲池专门负责 inode 对象的分配：

```
inode_cachep = kmem_cache_create("inode_cache", sizeof(struct inode), 0,
                                 SLAB_PANIC, init_once,NULL);
```

（2）kmem_cache_alloc()函数与 kmem_cache_free()函数

当需要分配一个拥有专用缓冲池的对象时，应该使用 kmem_cache_alloc()函数。下面的语句是从 inode_cache 缓冲池中获取一个空闲对象：

```
inode = (struct inode *) kmem_cache_alloc(inode_cachep, SLAB_KERNEL);
```

通过 kmem_cache_alloc()申请到的对象不用时通过 kmem_cache_free()函数释放，下面的语句将 inode 对象释放回 inode_cache 缓冲池：

```
kmem_cache_free(inode_cachep, inode);
```

（3）cache_grow()函数与 cache_reap()函数

kmem_cache_create()函数只是建立了所需的专用缓冲池的基础设施，但是此时缓冲池为空。slab 分配器的创建则要等到 kmem_cache_alloc()函数被调用时，却发现缓冲池中无空闲对象可供分配，此时通过 cache_grow()向伙伴系统申请一个 slab 空间，初始化 slab 分配器中的各个对象。每隔一定时间 cache_reap()函数被调用回收对象全空闲的 slab 分配器。

（4）kmalloc()函数与 kfree()函数

这两个函数分别用来向通用的缓冲池申请和释放内存。

3.2 进程地址空间的管理

每个进程都有自己独立的虚拟地址空间，从而保证多个进程可以同时运行且互不影响。每个进程的地址空间通过进程的页目录、页表实现与物理内存的映射。进程需要存储空间时并不是一开始就分配物理内存，而是分配一块虚拟空间，直到真正需要对物理内存进行操作时才通过“按需分页（Demand Paging）”机制分配物理内存。虚拟内存以页为基本单位，大小等同于物理页帧。

进程地址空间主要由 mm_struct 结构描述。该结构包含了进程地址空间的两个重要组成部分——进程的页目录及指向 vm_area_struct 结构的指针。

3.2.1 页表机制

此部分与硬件密切相关，我们以 i386 体系结构为例，描述如何把进程地址空间的线性地址转换成物理地址。i386 系列既支持分段机制，也支持分页机制，Linux 主要采用分页机制，常规情况下页的粒度为 4KB，页面可以映射到任一物理页帧。i386 下进程的线性地址为 32 位，分为以下三个部分：

① 页目录段。置于高 10 位，记录在页目录中的索引。

② 页表段。占据中间的 10 位，记录在页表中的索引。

③ 偏移段。占据低 12 位，表示在 4KB 的页帧中的偏移。

每个进程都有一个页目录，当进程运行时，寄存器 CR3 指向该页目录的基址。图 3-1 显示了从线性地址到物理地址的映射过程：

（1）从 CR3 取得页目录的基地址。页目录用一个物理页帧存储，用来保存页表的基址。每个数据项占 4 字节，因而页目录有 1024 个数据项。

（2）以线性地址的页目录段为索引，在页目录中找到页表的基址。页表也是用一个物理页帧存储，用来保存物理页帧号。每个数据项占 4 字节，因而页表有 1024 个数据项。

（3）以线性地址的页表段为索引，在页表中找相应的物理页帧号。

（4）物理页帧号加上线性地址的偏移段即得到了对应的物理地址。

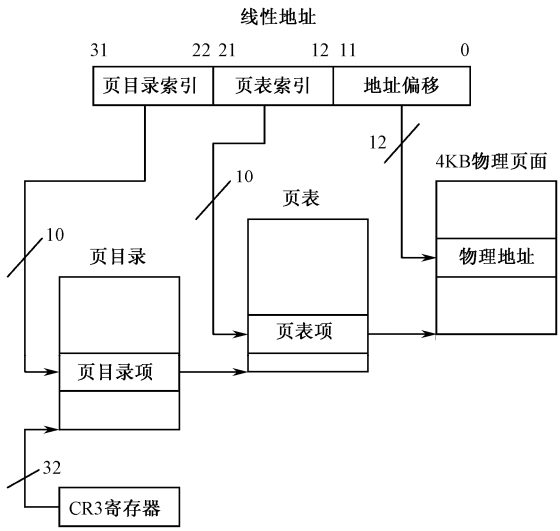

图 3-1　线性地址到物理地址的转换

因为物理页帧是 4KB 对齐的，所以页目录项和页表项都只需要 20 位保存物理地址的 12～31 位，还有 12 位可以用于控制或其他目的。4 个重要的标志位是：

① 存在位——表示项对应的物理页是否存在，一个典型的例子是当访问的线性地址所对应的物理信息还未装入物理内存时，其存在标志位为 0，将产生一个页面异常，导致申请一个物理页将信息装入并填充相应的页表项。

② 读/写位——表示是只读还是可写，起保护作用。比如存储代码的页面为只读，如果用户程序写入该页，将会引发写错误。

③ 访问位——表明该页是否被访问过。

④ 脏标志位——表明该页是否被写过。

采用两级页表的好处是节省了存储空间，比如页目录的第一项表示线性空间的前 4MB（1024×4KB）。如果该空间未使用，则只将页目录的第一项的存在位置为 0，表示不存在，而不需要分配一个页表再置各页表项为 0。如果不分级，则需要 1024 个页表项的存在位皆置为 0。两级页表对于 32 位机是合适的，对于 64 位机，往往采用级别更多的页表，甚至采用其他的页表组织结构。无论硬件的页表模型如何，Linux 采用的实际是 4 级页表模型，即页全局目录和页表，还有页上级目录层和页中间目录层。对于 32 位机，页上级目录层和页中间目录层实际不起作用。

处理器内部有称为 TLB 的专门缓冲空间用来加速地址转换过程。TLB 可看成（虚页，物理页）二元组的集合。进行地址转换时，首先查找 TLB 看是否有该线性地址项存在，如果有，则可以直接取得物理地址；如果没有，则再查找页表并更新 TLB。根据“局部性”原理，TLB 无疑将极大提高性能。

3.2.2 vm_area_struct结构

对于 i386，进程地址空间共有 4GB。其中前 3GB 可以为进程直接访问，称为用户地址空间，而后 1GB 空间为内核所用，称为内核地址空间，只有运行在核心态才能访问内核空间地址。目前绝大多数程序都不会使用全部的 3GB 空间。组成进程地址空间的各个部分性质不一定相同，如代码段、数据段显然不同，代码段只能读，而数据段可读写。Linux 的进程地址空间用一系列的 vm_area_struct（简称 VMA）结构描述不同的区间。

```
struct vm_area_struct {
        struct mm_struct * vm_mm;                  /*区间所属的 mm_struct 结构*/
        unsigned long vm_start;                    /*区间的起始地址*/
        unsigned long vm_end;                      /*区间的终结地址加一 */
        struct vm_area_struct *vm_next;            /*指向进程的下一个区间*/
        pgprot_t vm_page_prot;                     /*区间页帧的访问权限*/
        unsigned long vm_flags;                    /*区间的标志位*/
        struct rb_node vm_rb;                      /*代表该区间的红黑树结点*/
        ......
        struct vm_operations_struct * vm_ops;      /*区间的操作集合*/
        unsigned long vm_pgoff;                    /*在 vm_file 中的偏移*/
        struct file * vm_file;                     /*该区间映射的文件*/
        ......
};
```

每个 VMA 结构描述进程地址空间的一段连续区间，并且在该区间上的访问权限相同。各区间互不重叠，按线性地址的升序排列。查找某个线性地址属于哪个区间是内核频繁进行的操作，内核建立了一棵以 VMA 为结点的红黑树以保证搜索速度。

进程页面必定属于某个 VMA 结构，但是属于 VMA 结构的页面不一定在页表中有数据项。因为在需要用到该页面时才会申请物理页面并填充相应的页表项，在此过程中该页面的物理内容取决于其所属的 VMA 结构，见下面的页面异常处理。

3.2.3 进程地址空间的相关系统调用

内核在执行 exec()和 fork()系统调用时一般会产生全新的虚拟地址空间。前者会抛弃原来的地址空间并根据可执行文件内容生成相应的新的地址空间。后者会复制一份父进程的地址空间，其中父进程的 VMA 结构以及页目录、页表均会被复制，但是父进程页表项所标识的有效物理页帧并不会立即复制，而是由子进程和父进程共享，即父进程和子进程的同一页表项指向同一个物理页帧，并且页表项的相应标志位均标记为“只读”，这样当父进程或子进程试图写入该页帧时就会引发页面异常，处理过程见后面的论述。这种技术称为“写时复制”（Copy On Write），其优点如下：

（1）节省开销，有些在运行过程中不会被写到的页帧共享就足够了，没必要一个进程一份副本。

（2）要写入的页帧到要写的时候才复制一份，这是一种典型的“懒惰策略”，与一次性集中复制的效果相比，分散式复制即时响应效果更好。

系统调用 brk 用来改变堆的大小，增大进程地址空间。exit 结束进程并销毁进程地址空间。mmap 创建文件的内存映像，扩展进程地址空间，而 munmap 行为则与之相反。shmat 创建一个共享内存区，而 shmdt 则释放共享内存区。

3.2.4 页面异常的处理

导致页面异常的原因有两种：

（1）编程错误。可分为内核程序错误和用户程序错误。常见的用户程序错误有访问错误的地址空间和传递给系统调用错误的参数等，用户往往见到一个“Segmentation Fault”就是页面异常的结果。

（2）操作系统故意引发的异常。操作系统合理利用硬件机制在适当时间触发异常，使该异常处理程序被调用以达到预期目的。最常见的莫过于缺页，操作系统利用异常获得一个物理页帧后再重新执行产生异常的指令。

页面异常的处理程序是 do_page_fault()函数，该函数有两个参数，一个是指针，指向异常发生时寄存器上下文的地址；另一个是错误码，由三位二进制信息组成，第 0 位表示访问的物理页帧是否存在，第 1 位表示是写错误还是读错误或执行错误，第 2 位表示是程序运行在核心态还是用户态。do_page_fault()函数的执行过程如下：

① 首先得到导致异常发生的线性地址，对于 x86 该地址放在 CR2 寄存器中。

② 检查异常是否发生在中断或异常地址在核心空间，如是，则进行出错处理。

③ 检查该线性地址属于进程的某个 VMA 区间。如果不属于任何一个区间，则需要进一步检查该地址是否属于栈的合理可扩展区间。映射栈的区间可以向低地址方向扩展，该区间的 vm_flags 标志被设置成 VM_GROWSDOWN，即该区间的 vm_end 值固定，而 vm_start 的值可以减小。应用程序需要栈的大小往往需要到运行时刻才知道，并且是动态变化的，而区间的大小必须是 4KB 的整数倍。而且分配给栈区间的物理页帧只有到区间被删除时才释放，所以 Linux 采用了利用异常来扩展栈的机制。一旦用户态产生异常的线性地址正好位于栈区间的 vm_start 前面的合理位置，则调用 expand_stack()函数扩展该区间，通常是扩充一个页面，但此时还未分配物理页帧。至此，线性地址必属于某个区间。

④ 根据错误码的值确定下一个步骤。如果错误码的值表示为写错误，则检查该区间是

否允许写。若不允许，则进行出错处理；如果允许，就属于前面提到的写时复制。如果错误码的值表示为页面不存在，这就是所谓的按需分页。

下面分别讨论写时复制和按需分页的处理过程。

① 写时复制的处理过程是：首先改写对应页表项的访问标志位，表明其刚被访问过，以便在页面调度时该页面不会被优先考虑。如果该页帧目前只由一个进程单独使用，则只须把页表项置为可写。如果该页帧为多个进程共享，则申请一个新的物理页面并标记为可写，复制原来物理页面的内容，更改当前进程相应的页表项，同时原来的物理页帧的共享计数减一。

② 按需分页的处理过程是：首先确认产生页面不在物理内存的原因。一个原因是页面从未被进程访问，这种情况下，页表项的值全为 0。另一个原因是该页面被进程访问过，但是目前已被写到交换分区，这种情况下，页表项的存在标志位为 0，但其他位被用来记录该页面在交换分区中的信息。第一种情况下又要区分该页面是否是映射到一个文件。如果所属区间的 vm_ops->nopage 不为空，则表示该区间映射到一个文件并且 vm_ops->nopage 指向装入页面的函数，此时调用该函数装入该页面。如果 vm_ops 或 vm_ops->nopage 为空，则该调用 do_anonymous_page()申请一个页面。第二种情况下调用 do_swap_page()函数从交换分区调入该页面。

第 4 章　Linux的文件系统

Linux 为了支持多种不同的文件系统，引入了纯软件中间层 VFS，加入中间层无疑会使文件子系统的可扩展性、可维护性变得更好。本章先介绍 VFS 的运作原理，然后再介绍一个物理文件系统——EXT2，最后将介绍几个操作系统调用的实现。

4.1　VFS

4.1.1　VFS的作用

Linux 支持许多种文件系统，如 EXT2，VFAT，ISO9660 等。 VFS（Virtual File System）是内核软件层。它为用户空间的程序提供了大家熟悉的诸如 open()，read()之类的统一编程接口，同时隐藏了不同文件系统的差别，如图 4-1 所示。

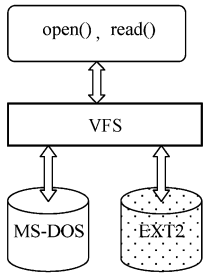

图 4-1　VFS 屏蔽普通物理文件系统的差别

4.1.2　进程描述符中与文件系统相关的成员

进程描述符中有下面两个与文件系统相关的成员：

```
struct fs_struct *fs;
struct files_struct *files;
```

其中，fs_struct 结构描述进程进行文件访问时用到的当前目录、根目录等信息。

```
struct fs_struct {
        ......
        struct dentry * root, * pwd, * altroot;
        struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
};
```

files_struct 结构描述被进程打开使用的文件信息，fd 成员指向一个 struct file *数组，数组中的已使用项指向某个正被打开的文件对象，数组中的索引称为文件描述符（File Descriptor），用户程序就是使用文件描述符对打开的文件进行操作。fd 开始时指向 fd_array

数组，进程打开的文件数超过 32 时，会重新分配一个更大的 struct file *数组并让 fd 指向它，前提当然是进程被允许打开更多的文件。

```
struct files_struct {
        ......
        struct file ** fd;
        ......
        struct file * fd_array[NR_OPEN_DEFAULT];/* NR_OPEN_DEFAULT 为 32*/
};
```

4.1.3 VFS的文件模型

VFS 的关键是，根据不同的文件系统抽象出了一个通用的文件模型。每个特定的文件系统都要把物理操作与通用文件模型对应起来。通用文件模型由 4 种数据对象组成。

1．文件对象

文件对象存储一个打开的文件和一个进程的关联信息。只要文件一直打开，这个对象就一直存在。文件对象用 struct file 结构描述：

```
struct file {
    struct list_head        f_list;
    struct dentry           *f_dentry;      /*指向与文件对象关联的 dentry 对象*/
    struct vfsmount         *f_vfsmnt;
    struct file_operations  *f_op;          /*文件对象的操作集合*/
    atomic_t                f_count;        /*引用计数*/
    unsigned int            f_flags;        /*使用 open()时设定的标志*/
    mode_t                  f_mode;         /*进程访问模式*/
    int                     f_error;
    loff_t                  f_pos;          /*对文件读/写操作的当前位置*/
    ......
 };
```

2．inode对象

inode 对象存储某个文件的管理信息，通常对应磁盘文件系统的文件控制块。文件在文件系统内有唯一的 inode 号。struct inode 结构的部分成员如下所述：

```
struct inode {
    struct hlist_node       i_hash;         /*用于 hash 表的指针*/
    struct list_head        i_list;         /*根据 inode 状态链入相关链表*/
    struct list_head        i_sb_list;      /*链入超级块的 inode 链表*/
    struct list_head        i_dentry;       /*inode 的 dentry 对象链表*/
    unsigned long           i_ino;          /*inode 号*/
    atomic_t                i_count;        /*使用计数*/
```

```
    umode_t                 i_mode;          /*表示文件类型及权限*/
    unsigned int            i_nlink;         /*硬链接的数目*/
    uid_t                   i_uid;           /*文件拥有者的用户 ID*/
    gid_t                   i_gid;           /*用户所在组的 ID*/
    dev_t                   i_rdev;          /*所在设备的设备号*/
    loff_t                  i_size;          /*文件大小*/
    struct timespec         i_atime;         /*最近一次的访问时间 */
    struct timespec         i_mtime;         /*最近一次的修改时间 */
    struct timespec         i_ctime;         /*文件创建时间*/
    ......
    struct inode_operations *i_op;           /*inode 对象操作集合*/
    struct file_operations  *i_fop;          /*文件对象操作集合*/
    struct super_block      *i_sb;           /*所属的超级块*/
    struct file_lock        *i_flock;
    struct address_space    *i_mapping;
    ......
};
```

3．dentry对象

dentry 对象主要是描述目录项及其相关联的 inode 信息。struct dentry 结构描述如下：

```
struct dentry {
    ......
    struct inode *d_inode;              /*  该 dentry 对象所属的 inode */
    struct dentry *d_parent;            /*  父目录 */
    struct qstr d_name;                 /*  文件名及附属信息*/
    ......
    struct dentry_operations *d_op;
    ......
};
```

4．超级块对象

超级块对象存储已安装文件系统的信息，通常对应磁盘文件系统的文件系统控制块。

```
struct super_block {
    struct list_head  s_list;           /*  将所有的超级块链接起来 */
    kdev_t            s_dev;            /*  所在设备号*/
    unsigned long     s_blocksize;      /*该文件系统磁盘块的大小(字节数)*/
    ......
    struct file_system_type*s_type;     /*所属的文件系统类型*/
    struct super_operations*s_op;
    ......
};
```

图 4-2 描述了各种对象之间的关系。inode 对象对应于一个文件，该文件一般在磁盘上，

inode 对象与文件之间的关系是一对一的。而一个真正的文件可能有多个文件名，比如硬链接（Hard Link），因此 dentry 对象与 inode 对象之间是多对一的关系。不同的进程可能打开同一个文件，但却不能用同一个文件对象来描述，因为操作标志、文件操作的位置等均可能不同，因而文件对象与 dentry 对象之间是多对一的关系。

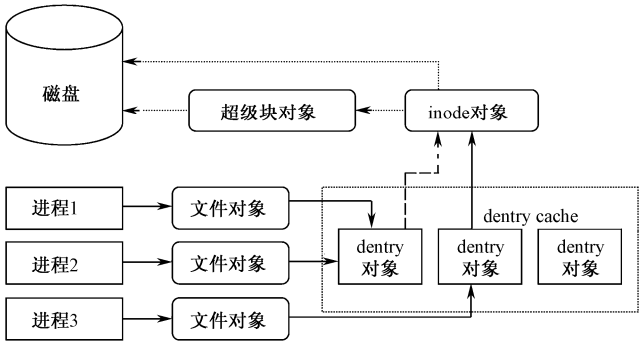

图 4-2　VFS 各种对象之间的联系

为了加快文件的查找速度，内核引入了缓冲机制。最近使用的 dentry 对象放在 dentry cache（简称 dcache）中，最近使用的 inode 对象放在 inode cache 中。在 cache 中的对象又以 Hash 表的形式将它们组织起来。

4.1.4　文件系统的注册与安装

1．文件系统的注册

每一种文件系统使用前必须先注册一个 file_system_type 对象。其成员如下:

```
struct file_system_type {
        const char *name;                                   /*文件系统类型名*/
        struct super_block *(*get_sb) (struct file_system_type *, int, const char *, void *);
        struct file_system_type * next;
       ......
};
```

get_sb 是一个函数指针，负责读取文件系统的超级块对象。每个文件系统都有自己的实现函数，如 EXT2 文件系统的具体项是 ext2_get_sb()。所有的文件系统类型通过 next 成员链接起来。

2．文件系统的安装

用户经常使用 mount 命令把一个分区安装到某个目录下。内核与之对应的系统调用是 sys_mount()。使用该函数一个重要步骤是，根据文件系统类型得到相应的 file_system_type 对象，再取得该分区的超级块对象。

4.1.5 各种对象的操作接口

1．面向对象的接口

前面已经提到，超级块对象、inode 对象、文件对象、dentry 对象构成了 VFS 框架。但它们又是如何与底层的物理文件系统打交道的呢？又是如何区分不同的文件系统的呢？Linux 采用面向对象的思想，每个对象都有一套函数操作集合，不同文件系统的对象其操作也就不同，VFS 只负责提供接口（操作种类集合）。下面以超级块对象为例来说明。

超级块对象有一个 struct super_operations *s_op 成员，struct super_operations 定义如下：

```
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);
        void (*read_inode) (struct inode *);
        void (*dirty_inode) (struct inode *);
        int (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        ......
};
```

当第一次安装（mount）一个 EXT2 分区时会调用 ext2_get_sb()，该函数有一条语句“sb->s_op = &ext2_sops”。ext2_sops 变量的成员值如下:

```
static struct super_operations ext2_sops = {
        .alloc_inode  = ext2_alloc_inode,
        .destroy_inode       = ext2_destroy_inode,
        .read_inode          = ext2_read_inode,
        .write_inode  = ext2_write_inode,
        .delete_inode = ext2_delete_inode,
        .put_super           = ext2_put_super,
        .write_super  = ext2_write_super,
        ......
};
```

read_inode 等函数指针初始化 ext2_read_inode()之类特定于 EXT2 文件系统的函数入口，而当第一次安装（mount）一个 FAT 分区时会调用 fat_read_super()，该函数有一条语句“sb->s_op = &fat_sops”。fat_sops 变量的成员值如下:

```
static struct super_operations fat_sops = {
        .alloc_inode    = fat_alloc_inode,
```

```
        .destroy_inode= fat_destroy_inode,
        .write_inode   = fat_write_inode,
        .delete_inode  = fat_delete_inode,
        .put_super     = fat_put_super,
        ......
};
```

从面向对象的角度来描述上面的概念，可以认为 super_operations 就是一个抽象类，它只提供接口但并没有实现这些接口，而 ext2_sops，fat_sops 则是具体类，实现相应的接口。与超级块对象类似，每个 inode 对象包含一个 struct inode_operations *i_op，每个文件对象包含一个 struct file_operations *f_op，每个 dentry 对象包含一个 struct dentry_operations *d_op。每个对象的具体操作取决于该对象所属的文件系统类型。下面介绍各个主要接口。

2．struct super_operations

alloc_inode——分配一个 inode 对象。

destroy_inode——注销一个 inode 对象。

read_inode——该函数从已 mount 的文件系统读入一个 inode 信息。该 inode 对象的 inode 号已事先被初始化。

dirty_inode——当 inode 标记为脏时调用。

write_inode——把 inode 信息写入磁盘时使用该函数。

put_inode——从 inode cache 移去 inode 对象时，调用该函数。

drop_inode——当 inode->i_count 变为 0 时调用。

delete_inode——删除 inode 时调用。

put_super——卸载时，VFS 释放超级块时调用。

write_super——希望把 VFS 超级块写入磁盘时调用。

3．struct inode_operations

```
struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
        struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        ......
};
```

注意目录和文件在 VFS 中都用 inode 对象表示，而目录项则由 dentry 对象表示。下面介绍前两个函数，以便有助于对整体有所了解。

① create——第 1 个参数必须是目录型文件对应的 inode。create()在该目录下创建一个文件，文件名事先被置入第 2 个参数中。第 3 个参数是文件的创建类型。

② lookup——第 1 个参数必须是目录型文件对应的 inode。lookup()在该目录下查找一个文件，文件名事先被置入第 2 个参数 dentry 中。查找到的信息填入 dentry 中并返回。

4．struct file_operations

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        ......
}
```

这些操作与系统调用接口非常相似，因此不过多介绍。

5．struct dentry_operations

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, struct nameidata *);
        int (*d_hash) (struct dentry *, struct qstr *);
        int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
        ......
};
```

① d_revalidate——检查 dentry cache 中的 dentry 是否是最新的。

② d_hash——计算文件名的 Hash 值。第 1 个参数是父目录，第 2 个参数包含了预设的文件名。

③ d_compare——比较一个目录下的两个文件名是否相等。如 MS-DOS 文件系统不区分大小写，而 UNIX 文件系统则不然。

4.2 EXT2 文件系统

EXT2 文件系统是 Linux 的原生文件系统之一，它支持传统 UNIX 文件的语义及一些高级特性，在性能和健壮性方面都表现不错。EXT2 的衍生版本 EXT3，在 EXT2 的基础上增加了日志功能，以满足企业级应用对日志文件系统的需求。

4.2.1 EXT2 在磁盘上的物理布局

EXT2 分区的第一个磁盘块用于引导，其余的部分被分成组，见图 4-3。所有的组大小相同且顺序存放，所以由组的序号可以确定组在磁盘上的位置。

每个组由如下 6 部分组成：（1）文件系统的超级块；（2）所有组的描述符；（3）数据块

的位图；（4）inode 位图；（5）inode 表；（6）数据块。

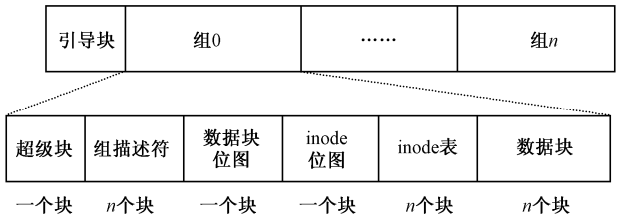

图 4-3　EXT2 分区的物理布局

早期版本的 EXT2 文件系统每个组都有一份文件系统超级块和所有组描述信息的副本，正常情况下，内核只使用第 0 组的超级块和所有组的描述信息，并在合适时机让各个组的副本一致。当组 0 的副本遭到损坏时，便可根据其他组的副本恢复，从而加强文件系统的可靠性。当文件系统很大时，将浪费大量空间备份超级块和组描述符，EXT2 文件系统随后的版本支持稀疏超级（Sparse Super）特性，当组号是 0、1 或 3、5、7 三个数的幂时，该组才需要备份超级块和组描述符。

而块的位图、inode 表、inode 位图、数据块则专属于组。块位图占用一个块，位图的每一位顺序对应组中的一个块，0 表示块可用，1 表示已被用。如果块的大小为 1KB，则每个组的大小为 8196 块。所有的文件及目录都需要用 inode 结构表示，inode 结构大小固定，依次存放在 inode 表中。inode 位图用来表示对应的 inode 表的空间是否已被占用。

4.2.2　主要的数据结构及其基本操作

1．超级块

超级块的信息可用 ext2_super_block 结构表示，下面列出了一些讨论中用到的成员。

```
struct ext2_super_block {
        __le32  s_inodes_count;         /* inode 的总数 */
        __le32  s_blocks_count;         /* 块的总数 */
        __le32  s_r_blocks_count;       /* 保留块的总数 */
        __le32  s_free_blocks_count;    /* 空闲块的总数 */
        __le32  s_free_inodes_count;    /* 空闲 inode 的总数 */
        __le32  s_first_data_block;     /* 第一个数据块 */
        __le32  s_log_block_size;       /* 块的大小 */
        __le32  s_log_frag_size;        /* 碎片的大小 */
        __le32  s_blocks_per_group;     /* 每组的块数 */
        __le32  s_frags_per_group;      /* 每组的碎片数 */
        __le32  s_inodes_per_group;     /* 每组的 inode 数 */
        __le32  s_mtime;                /* 安装的时间 */
        __le32  s_wtime;                /* 写的时间 */
        __le16  s_mnt_count;            /* 安装的次数 */
        ......
};
```

2．组描述符

每个组都有自己的描述符，内核用结构 ext2_group_desc 描述。下面列出它的主要成员:

```
struct ext2_group_desc
{
        __le32   bg_block_bitmap;          /* 本组数据块位图所在的块号 */
        __le32   bg_inode_bitmap;          /* 本组 inode 位图所在的块号 */
        __le32   bg_inode_table;           /* 本组 inode 表的起始块号 */
        __le16   bg_free_blocks_count;     /* 组中空闲块的数目 */
        __le16   bg_free_inodes_count;     /* 组中空闲 inode 的数目 */
        __le16   bg_used_dirs_count;       /* 组中目录的数目 */
        __le16   bg_pad;                   /* 未用 */
        __le32   bg_reserved[3];           /* 未用 */
};
```

3．inode

inode 就是 EXT2 文件系统的文件控制块，EXT2 文件系统所有的 inode 大小均为 128B，每个 inode 在 inode 表放的位置就是该 inode 隐含的 inode 号。内核用结构 ext2_inode 描述 inode，主要成员如下：

```
struct ext2_inode {
        __le16      i_mode;
        __le16      i_uid;          /* 拥有者的用户 ID */
        __le32      i_size;         /* 文件大小 */
        __le32      i_atime;        /* 最近一次访问时间 */
        __le32      i_ctime;        /* 创建时间 */
        __le32      i_mtime;        /* 最近一次修改时间 */
        __le32      i_dtime;        /* 最近一次不用时间 */
        __le16      i_gid;          /* 文件的组 ID */
        __le16      i_links_count;  /* 链接数 */
        __le32      i_blocks;       /* 分配给该文件的磁盘块的数目 */
        __le32      i_flags;        /* 文件标志 */
        ......
        __le32      i_block[EXT2_N_BLOCKS]; /* 指向磁盘块的指针 */
        ......
};
```

i_mode 包含了文件的类型信息和文件的访问权限。文件的类型有常规文件、目录、字符设备、块设备等。

i_block[EXT2_N_BLOCKS] 一般用于放置文件的数据所在的磁盘块编号，EXT2_N_BLOCKS 的默认值为 15。

i_block[15]数组主要是支持常规文件，因为数据在磁盘上并不一定连续，需要保存各个磁盘块号。它的前 12 项可看成一级指针，直接存放文件数据所在的磁盘块号。数组的第 13 项是

个二级指针，指向的磁盘块并不包含文件的数据，而是一系列的一级指针，这些一级指针才用来指向磁盘块。假设磁盘块大小是 1KB，每个磁盘块号占 4 字节，则第 13 项可表达文件的大小是 256KB。而数组的第 14、15 项分别是三级指针和四级指针，访问数据方式可由第 13 项类推。这种方法保证了对大量的小文件访问效率高，同时又支持大文件，如图 4-4 所示。

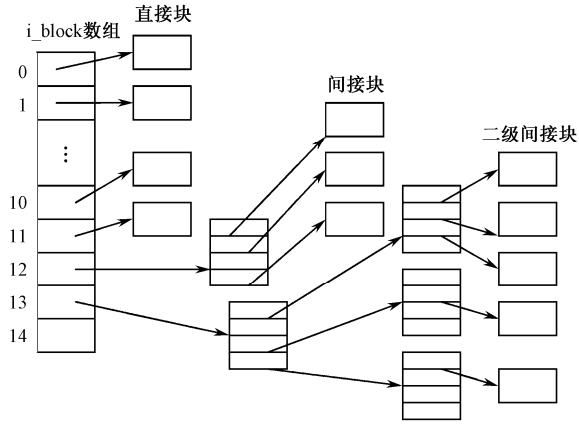

图 4-4　EXT2 文件数据块组织示意图

设备文件用 ext2_inode 就足以包含所有信息，不需要另外的数据块，而常规文件、目录有所不同，一般情况下总需要额外的磁盘块来存储数据。

目录的数据块包含了所有属于这个目录的文件信息。它的数据项类型是 ext2_dir_entry_2 结构，用来描述属于这个目录的文件。原型声明如下:

```
struct ext2_dir_entry_2 {
        __le32      inode;                  /* inode 号  */
        __le16      rec_len;                /*  本项所占的长度  */
        __u8 name_len;                      /*  文件名长度  */
        __u8 file_type;                     /*  文件类型  */
        char  name[EXT2_NAME_LEN];          /*  文件名  */
};
```

目录的各项在数据块中依次放置，可以看成是一个变长数组，每一项的起始地址偏移 rec_len 后即是下一项的起始地址。当删除一项时，被删除项的 rec_len 加入前一项的 rec_len 即可。如果要增加一项，则顺序查找已有项，如果某项的剩余空间足以容纳新项，则利用该空间放置，找不到时需要分配新块。

4．一些操作

如果知道文件的 inode 号，如何找到该文件信息呢？因为每组磁盘块的 inode 数目固定，所以很容易计算出该文件属于哪个组并且得到 inode 在组中 inode 表的下标，继而可得到 ext2_inode 信息。

再介绍如何查找一个文件，以/root/temp.c 为例。由于根目录的 inode 号总为 2，因此可以得到根目录的 ext2_inode 信息，再从 i_block 指向的数据块查找是否有 ext2_dir_entry_2 项的名字等于 root，如果有，就能得到了 root 目录的 inode 号。重复上述过程，则可以判定是

否存在/root/temp.c。

最后，介绍如何读取文件某个位置的数据。由于给定的位置是相对于文件而不是相对于磁盘的，因此需要根据该位置计算出它在 i_block 中的下标，这样才能得到在磁盘上的位置。当然，因为多级指针的介入而稍显复杂。

4.2.3 磁盘块的分配与释放

磁盘块的释放主要工作是修改块位图和涉及块的统计变量。

磁盘块的分配稍为复杂。EXT2 采用了预分配策略，当申请一个块时，系统尽量预分配 8 个连续块，这样下一次申请时，分配算法先查找是否存在预分配块，如果有，存在分配即可。如果不行，则试图在附近 32 个块的范围内分配。若还不行，则在本组内向前找 8 个连续空闲的块。若还不满足，则任何空闲的块均可以被分配。若还不满足，则到其他的组中去寻找。这样做能有效地减少文件的访问时间。

4.3 主要文件系统的系统调用处理流程

文件系统的操作接口 open()，read()，write()等看似简单，但内部实现却比较复杂，下面介绍 open()，read()的实现，以见一斑。

4.3.1 文件的open操作

open()函数最终会调用内核的 sys_open()函数，该函数的第 1 个参数是打开文件的路径名，第 2 个参数是对文件的访问标志，可以是读、写、创建等标志位的组合。该函数的执行过程如下：

（1）调用 getname()将用户空间的文件名复制到内核空间供后面的步骤使用。

（2）调用 get_unused_fd()在 current->files->fd 所指向的文件对象指针数组中查找一个未使用的索引（文件描述符），将该值存储在局部变量 fd 中。current 表示当前进程描述符。

（3）调用 filp_open()函数，该函数的工作主要分成两步：

第一步：调用 open_namei()函数，该函数的功能是找到目标结点（可以是文件、目录）所对应的 dentry 对象。与 dentry 对象相对应的 inode 对象此时也应该在物理内存中。open_namei()的主体就是调用 path_lookup()填充一个 nameidata 结构。path_lookup()函数的执行过程如下：

① 确定从哪一个 dentry 对象出发进行路径解析。根据指定文件路径名是相对路径还是绝对路径，从 current->fs 中得到相应的 dentry 对象。

② 调用 link_path_walk()函数进行路径解析，一系列文件系统操作均会调用该函数。

以 cs371/project/document.html 为例，目前已经得到了当前目录的 dentry 对象，现在要得到子目录 cs371 的 dentry 对象，再通过 cs371 的 dentry 对象得到 project 的 dentry 对象，依次下去，实际上是个循环过程。每一次循环都得到一个 dentry 对象，该对象对应文件路径的一个子路径，下面是每次循环的具体过程：

① 如果子路径是“.”表示当前目录，则应该进行下一次循环。

② 如果子路径是“..”表示父目录，则要看当前 dentry 对象的情况。若当前 dentry 对象已经是本进程的根目录，这时应保持不变，直接进行下一次循环；若当前 dentry 对象与父目

录在同一设备上，则 dentry 对象的 d_parent 成员即为所求，d_parent 一定在内存中。还有一种情况是当前 dentry 对象为所在设备的根结点，它的上一层必然是另一个设备，此时应把 dentry 对象改成安装点的 dentry 对象，再从头执行②。

③ 调用 do_lookup()找到或建立子路径的 dentry 对象。

__d_lookup ()函数查找子路径对应的 dentry 是否已经在 dentry cache（dcache）中，如果有，则返回。

此时应该到磁盘上查找信息，调用 real_lookup()函数，该函数先通过 slab 分配器申请一个空闲 dentry 对象，用来存放即将查找的信息，然后调用父目录 inode 对象的 i_op->lookup()方法，lookup()方法是特定于文件系统的，下面以 EXT2 文件系统为例说明大致流程。

EXT2 的 lookup 方法是 ext2_lookup()函数，它首先从磁盘读入 dentry 对象信息，包括 dentry 对象对应的 inode 号；然后再查找相应的 inode 对象是否在 inode cache 中，如果不在，则申请一个空闲的 inode 对象，再利用超级块对象的 s_op->read_inode()方法从磁盘读入相关信息，EXT2 的 read_inode 方法是 ext2_read_inode ()函数；最后，让 dentry 对象的 d_inode 成员指向该 inode 对象。

④ 已经找到子路径的 dentry 对象，但还有两种情况值得考虑，一是该 dentry 对象是一个安装点，这种情况下要推进到所安装设备的根结点；二是该 dentry 对象是一个软链接（Soft Link），此时要推进到链接目标。

第二步：调用 dentry_open()函数，该函数申请获得一个空闲的文件对象 f，然后初始化该对象，其中包括使 f->f_dentry 指向已获得的 dentry 对象，f->f_op 被赋值为 inode 对象的 i_fop 成员，然后将调用 f->f_op->open()。

（4）调用 fd_install()函数，通过下面这条语句将文件对象装入当前进程的打开文件表：

current->files->fd[fd] = file;

（5）返回文件描述符 fd。

4.3.2 文件的read操作

1．页缓冲（page cache）

Linux 在读/写文件时采用了页缓冲的机制，采用该机制能够有效地减少 I/O 操作的次数。尽管文件在磁盘上以磁盘块为基本单位存放，但在内核中文件被看成由页面组成，对文件的读写首先要经过页缓冲。读操作首先到页缓冲中查找页面是否存在，如果存在则将结果返回。如果没找到，则发出 I/O 请求。写操作并不立即发出 I/O 请求，而是写入页缓冲，在后面的某个时刻才会启动 I/O 操作，如图 4-5 所示。

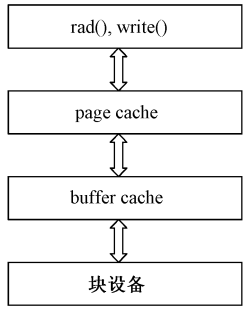

图 4-5 在文件操作中的页缓冲

每个页面由 struct page 结构描述。每个文件的所有页面由一个 struct address_space 结构管理，struct inode 结构中的 i_mapping 成员就起这个作用。当对属于文件的页面进行操作时，就需要调用 address_space_operations 的相应方法。

```
struct address_space {
    struct inode        *host;                  /*  该空间的拥有者*/
    struct radix_tree_root    page_tree;        /*  基树的根  */
    ......
    struct address_space_operations *a_ops;
    ......
};
struct address_space_operations {
    int (*writepage)(struct page *page, struct writeback_control *wbc);
    int (*readpage)(struct file *, struct page *);
    ......
};
```

为了快速查找文件的页是否在 page cache 中，同一个 address_space 在 page cache 的页面组成一棵基树，基树根据当前访问过的文件最大偏移量来生成与调整，对于大小不超过 256KB 的小文件，相应的基树深度仅为 1。

2．read()的实现

read()在内核中的对应函数为 sys_read()，该函数的执行过程如下：

（1）根据文件描述符调用函数 fget_light()，找到相应的文件对象 file。

（2）调用 file_pos_read()得到当前读指针。

（3）调用 vfs_read()函数执行读的操作。

① 检查 file->f_mode 是否允许读。

② 调用 rw_verify_area()检查将读的文件区间是否被锁住，如果只有读锁，则可以继续。如果是互斥的写锁，根据写锁设置进程阻塞或者返回一个错误码。

③ 调用 file->f_op->read()函数执行读操作，如果 read 函数不存在，则调用 file->f_op->aio_read 读数据。对于大部分文件系统，struct file_operations 的 read 成员实际是指向 generic_file_read()函数。该函数根据文件位置和要读出的长度确定相应的页面，然后再检查该页面是否在 page cache 中存在。如果不存在，就要调用 inode 结点的 i_mapping->a_ops->readpage()方法将其从磁盘读入，不同磁盘文件系统的 readpage 方法不同，EXT2 文件系统相应的函数为 ext2_readpage()。另外需指出的是，为了提高性能，读操作采用了预读机制以减少 I/O 操作的次数。

（4）调用 file_pos_write()更新当前读指针。

（5）返回实际读到的字节数。

第 5 章　Linux的设备管理

Linux 沿用了 UNIX 处理设备的做法，用户可以像处理常规文件一样来操作设备。本章将先介绍了设备文件的概念及相关的数据结构，最后通过 open()与 read()的实现描述块设备管理是如何与 VFS 框架结合的。

5.1　设备文件的概念

传统的 UNIX 系统均把设备当成文件来处理，因而可以用 read()/write()对设备进行操作。设备文件一般在/dev 目录下，如/dev/fd0 表示软盘，/dev/hda1 表示第一个硬盘的第一个分区。

Linux 下的设备分为三类:

（1）块设备。一次 I/O 操作以固定大小的数据块为单位，且可随机存取。

（2）字符设备。一次 I/O 操作存取数据量不固定，只能顺序存取。

（3）网卡。网卡是特殊处理的，它没有对应的设备文件。

设备文件除文件名之外，还有三个主要属性：

（1）类型，表明是字符设备还是块设备。

（2）主设备号，主设备号相同的设备，被同一设备驱动程序处理。

（3）从设备号，用来指明具体的设备。

例如，/dev/hda1，/dev/hda2 都是块设备文件，主设备号均为 3，从设备号则分别为 1 和 2。

设备文件的生成由 mknod()系统调用完成，它的参数是上面提到的三个属性。Linux 安装完成之后已经在/dev 目录下生成了绝大多数可能要用到的设备文件，尽管很多真正的设备尚未安装。

5.2　设备模型基础

内核对象是 Linux 2.6 内核引入的新的设备管理机制，在内核中用 kobject 结构表示。kobject 是一种简单的数据类型，它一般内嵌在更为复杂的的数据类型中。kobject 提供基本的对象管理，是构成 Linux 2.6 设备模型的核心结构，它与 sysfs 文件系统紧密关联，每个在内核中注册的 kobject 对象都对应于 sysfs 文件系统中的一个目录，目录下的文件往往是对象的属性。使用这个结构，所有设备在底层都具有统一的接口。内核对象结构定义如下：

```
struct kobject {
        char            * k_name;               /*设备名称，名字短时指向 name 数组*/
        char      name[KOBJ_NAME_LEN];
        struct kref           kref;             /*对象引用计数*/
        struct list_head      entry;            /*挂接到所在内核对象集合中去的单元*/
        struct kobject        * parent;         /*指向父对象的指针*/
        struct kset           * kset;           /*所属内核对象集合的指针*/
```

```
    struct kobj_type    * ktype;        /*指向其对象类型描述符的指针*/
    struct dentry       * dentry;       /*在 sysfs 文件系统中对应的文件结点路径指针*/
};
```

下面是一些内核对象操作接口：

```
int kobject_register(struct kobject *);
void kobject_unregister(struct kobject *);
struct kobject * kobject_get(struct kobject *);
void kobject_put(struct kobject *);
void kobject_init(struct kobject *);
int kobject_add(struct kobject *);
```

函数 kobject_register()会注册一个内核对象，把内核对象加入一个内核对象集合，并在 sysfs 文件系统中创建对应该内核对象的目录。kobject_unregister()则删除内核对象，同时对象计数减一。

函数 kobject_get()将 kobject 的引用计数加一，而函数 kobject_put()将 kobject 的引用计数减一。

函数 kobject_init()和 kobject_add()用来初始化和添加一个 kobject。调用 kobject_init()后再调用 kobject_add()，同调用 kobject_register()有相同的功能，即注册一个 kobject。

5.3 相关数据结构

5.3.1 字符设备管理

字符设备管理的主要数据结构如下：

```
struct cdev {
    struct kobject kobj;                /*内嵌的内核对象  */
    struct module *owner;               /*所属的模块*/
    struct file_operations *ops;        /*设备操作集合*/
    struct list_head list;              /*设备的 inode 链表头*/
    dev_t dev;                          /*设备号*/
    unsigned int count;                 /*分配的设备号数目*/
};
static struct char_device_struct {
    struct char_device_struct *next;    /*链接哈希值相同的项*/
    unsigned int major;                 /*主设备号*/
    unsigned int baseminor;             /*第一个从设备号*/
    int minorct;                        /*从设备号区间的大小*/
    const char *name;                   /*设备名*/
    struct file_operations *fops;       /*未使用*/
```

```
    struct cdev *cdev;                  /*相关联的 cdev 对象*/
} *chrdevs[MAX_PROBE_HASH];
```

每个字符设备驱动都有一个 cdev 结构，为了追踪设备号的使用情况，系统使用哈希数组 chrdevs 记录已经分配的主设备号和从设备号，char_device_struct 结构描述被分配出去的设备号区间。设备在系统中注册是使用函数 register_chrdev()完成，其原型如下：

int register_chrdev(unsigned int major, const char * name,struct file_operations *fops);

major 是用户期望的主设备号，如果值为 0，则由系统选择一个空闲的主设备号；name 是设备名；fops 是字符设备特定的操作集合。register_chrdev()分配一个 cdev 对象和一个 char_device_struct 对象并将它们初始化。设备注册后就可以使用该设备，open 操作打开字符设备文件时，文件对象的操作集合就被初始化为该设备特定的操作集合，从而和普通的文件区分开来。当设备不再使用时，文件可以通过 unregister_chrdev()函数注销。

5.3.2 块设备管理

块设备的管理比字符设备的管理要复杂得多，在此只能简单介绍如下。每个块设备由 gendisk 结构描述，其定义如下：

```
struct gendisk {
    int major;                              /*  主设备号  */
    int first_minor;                        /*  第一个从设备号  */
    int minors;                             /*  设备最大的从设备数  */
    char disk_name[32];                     /*  设备名  */
    struct hd_struct **part;                /*  分区数组  */
    struct block_device_operations *fops;   /*  块设备的操作集合  */
    struct request_queue *queue;            /*  请求队列  */
    void *private_data;                     /*  私有数据  */
    sector_t capacity;                      /*  块设备的容量  */
    ……
    struct kobject kobj;                    /*  内嵌的 kobject 对象  */
    ……
};
```

要使用一个块设备，用 register_blkdev 注册设备号，再使用 alloc_disk()函数分配一个 gendisk 对象并对该对象进行初始化，其中包括请求队列对象和块设备操作集合的初始化，然后注册一个中断处理函数，最后调用 add_disk()注册 kobject 对象，激活该设备。

当打开某个块设备文件时，系统将创建一个 block_device 对象，结构定义如下：

```
struct block_device {
    dev_t       bd_dev;                 /*  设备号  */
    struct inode    *bd_inode;          /*  对象所对应的 inode 对象  */
    int         bd_openers;             /*  打开的次数  */
    ……
    struct gendisk    * bd_disk;        /*所对应的 gendisk 对象  */
```

```
……
};
```

5.3.3 buffer

文件系统对块设备的操作是以块为基本单位，而磁盘的物理基本单位是扇区，块的大小不少于一个扇区但不会超过页面的大小。每读入一个块时，并不立即把块的数据送到应用程序的缓冲区，而是先在内核申请一个同等大小的内存块称为 buffer，将块读入，然后再写入程序缓冲区。这样，下一次访问该块时就不必进行磁盘操作，从而提高了性能。对块的写操作同样要经过 buffer，每个 buffer 由 buffer_head 结构描述，该结构定义如下：

```
struct buffer_head {
    unsigned long b_state;              /* 状态位图 */
    struct buffer_head *b_this_page;    /* 同属一个页面的 buffer 链表 */
    struct page *b_page;                /* buffer 所属的页面 */
    atomic_t b_count;                   /* 使用者计数 */
    u32 b_size;                         /* 块的大小 */
    sector_t b_blocknr;                 /* 块号 */
    char *b_data;                       /* buffer 所在的位置 */
    struct block_device *b_bdev;        /* 所属的块设备 */
    ……
};
```

每个 buffer 由设备和块号唯一确定。若块的大小为 1KB，物理页帧的大小为 4KB，则一个物理页帧可以容纳 4 个 buffer，图 5-1 反映了这种关系。

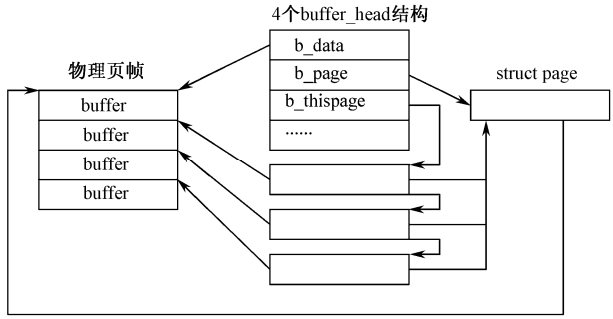

图 5-1　同属于页面的 buffer_head 之间的关系

下面介绍 page cache 和 buffer cache 的区别。早于 2.4.10 版本的 Linux 内核，buffer cache 起着非常重要的作用，使用 buffer cache 的场合主要是读取块设备文件的数据和文件系统的元数据（如 EXT2 的超级块信息、块位图等），page cache 主要用来存放普通文件的数据。在 2.6 内核中，文件的数据几乎都由 page cache 来组织，文件可以是普通文件也可以是特殊文件，如块设备文件等，文件数据并不一定在物理上连续，读入 page cache 的操作必然以块为基本单位完成，所以 page cache 是依赖于 buffer 的。

5.3.4 设备请求队列和I/O调度算法

每个块设备都维护一个请求队列，该队列的每一个成员称为请求。一个请求实际上主要包含了两项信息，一是若干物理上连续的磁盘块，二是操作类型是读还是写。磁盘驱动逐个完成请求，直到请求队列为空。每当文件系统产生一个新的请求，都要将其加入设备请求队列，这个过程有两个优化措施：

① 加入之前，先要检查已有的块设备操作请求和现在的请求是否在物理上相邻并且操作行为一致。如果是，则将该操作合并入已有的操作请求而无须生成新的操作请求。

② 如果不能合并，则生成一个新的块设备操作请求，由 I/O 调度器加入设备请求队列的合适位置。

I/O 调度器采用的算法是影响系统性能的关键因素，令人遗憾的是，没有一种 I/O 调度算法能够适合所有的 I/O 负载情况。Linux 2.6 支持 4 种 I/O 调度算法，它们分别是预期 I/O 调度算法、期限 I/O 调度算法、完全公平队列 I/O 调度算法和 NOOP I/O 调度算法，默认情况下是预期 I/O 调度算法。在 Linux 启动时，通过设置内核参数可以选用 4 种算法中的一种，在运行时刻，系统管理员也可以通过 sysfs 文件改变某个块设备所采用的 I/O 调度算法。

5.4 块设备文件的open和read操作

读取块设备上的数据有两种形式，一种是把它看成以某种文件系统组织起来的各个文件的集合来读取，另一种则是把块设备整体看成一个文件来访问。已经介绍过前一种形式，下面我们介绍后一种形式。无论哪一种形式，最终都要归结为对块的操作。

5.4.1 块设备驱动程序组成

从图 5-2 可以看出，块设备驱动程序分为两部分：① 与 VFS 的接口层；② 真正对设备（一般是磁盘控制器）操作的部分，有操作请求以后会触发这部分。

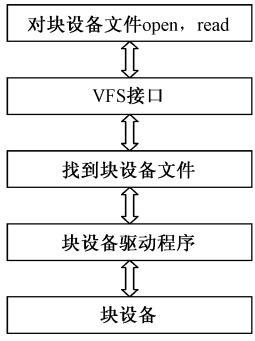

图 5-2 块设备文件的操作流程

5.4.2 open函数

该函数的处理与第 4 章介绍的流程一样，我们主要介绍该流程中如果设备作为一种特殊文件是如何处理的。

调用 open_namei()函数过程中需要填充一个与设备文件相对应的 inode 对象的各成员项，这将会调用 init_special_inode()函数。

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {        /*为字符设备文件*/
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = rdev;
    } else if (S_ISBLK(mode)) {  /*为块设备文件*/
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = rdev;
    }
    ……
}
```

对设备的操作集合为 def_blk_fops，与普通文件系统的操作集合不同，根据第 4 章的介绍，该操作集合的 open 方法，即 blkdev_open()函数被调用。此函数的作用是取得该设备对应的 block_device 对象和 gendisk 对象，对两个对象进行初始化，如有必要，还要对该设备进行初始化。

```
struct file_operations def_blk_fops = {
    .open        = blkdev_open,
    ......
    .read        = generic_file_read,
    .write       = blkdev_file_write,
    ......
};
```

5.4.3 read函数

从 def_blk_fops 的成员列表可以看到，对于特殊文件类型块设备文件的读操作，也采用了通用的文件读函数 generic_file_read()，该函数在第 4 章已经介绍。

第 6 章　中断、异常及系统调用

本章将基于 80x86 分别介绍中断、异常及系统调用的处理过程。中断通常分为同步中断和异步中断。同步中断由 CPU 产生，通常情况下是执行指令时遇上异常情况，常见的同步中断有被零除、溢出及页面异常等。产生同步中断的情况中有一种是使用 int 指令，Linux 使用该指令来实现系统调用。异步中断又分为可屏蔽的和不可屏蔽的两类，由一些硬件设备产生，可以在指令执行的任意时刻产生，I/O 设备和定时器产生的中断就属于异步中断。

Intel 处理器手册把同步中断称为异常，把异步中断直接称为中断，我们在此采用这种说法。

6.1　中断和异常的基本知识

中断和异常的发生都将导致核心态代码的运行。中断既可能发生在用户态，也可能发生在核心态，并且中断的嵌套发生往往是允许的。相比之下，异常大部分发生在用户态，但是页面异常可能发生在核心态，这往往是因为在核心态要访问进程地址空间时该页面又不在内存中。可以看到，即使在核心态可能发生页面异常，但异常最多可能有两层嵌套，绝大多数情况下只有一级异常。异常的处理过程中可能产生中断，反之则不可能。

每个中断和异常都有一个向量号，该号的值在 0～255 之间，该值是中断和异常在中断向量表 IDT（Interrupt Descriptor Table）中的索引。每个中断和异常均有其相应的处理函数，中断和异常在使用前必须在 IDT 中注册信息以保证发生中断和异常时能找到相应的处理函数。IDT 表项还记录了一些其他信息，用于安全检查和防止非法进入核心态。IDT 在系统初始化时创建。

IDT 中向量号的使用情况如下：

① 异常与非屏蔽中断使用 0～31。

② 可屏蔽中断使用 32～47，见 6.4 节。

③ Linux 内核用第 128（0x80）号中断实现系统调用。

④ 剩余的编号，主要用于外部中断、高级可编程中断控制器（APIC）。

人们非常熟悉中断和异常发生与退出的处理，一句话就是发生时保存现场，退出时恢复现场。这个工作是由软件和硬件共同完成的。值得指出的是，保存在核心栈的程序计数器 eip 的值取决于具体情况。一般情况下，eip 保存下一条指令的地址，但对于页面异常，保存的是产生异常的这条指令的地址，而不是下一条指令的地址，因为这很可能是由缺页所导致的，异常处理将调入页面，然后重新执行这条指令，见第 3 章。

6.2　异常处理函数

异常处理的一般流程如下：

① 将大多数寄存器的值保存到核心栈上。

② 调用相应的处理函数。

③ 跳转到 ret_from_exception 标号处退出。

大多数异常处理函数的过程如下：

① 硬件错误码和异常向量号存入当前进程。

② 给当前进程发出一个相应的信号。

下面的代码说明了该过程。

```
struct task_struct *tsk = current;              /*current 为当前进程*/
tsk->thread.error_code = error_code;            /*error_code 为硬件错误码*/
tsk->thread.trap_no = trapnr;                   /* trapnr 为异常向量号*/

/* force_sig_info() force_sig() 均用来向进程发出信号*/
if (info)
        force_sig_info(signr, info, tsk);
else
        force_sig(signr, tsk);
```

当从 ret_from_exception 处返回用户空间时，将会检查进程是否有信号等待处理。如果有，则看进程是否有自己的信号处理函数。有则调用它，否则采用内核对信号的默认处理，一般是将该进程杀死。

但是，有些异常的处理过程并不如上面的情况。典型的例子是页面异常，该异常是实现按需分配页面的硬件支持机制，它的处理函数最终会调用 do_page_fault()函数，参见第 3 章的相关内容。

6.3 系统调用

众所周知，用户程序不能对硬件直接操作。操作系统为用户态进程提供了统一的接口——系统调用，通过该接口，用户态进程可以切换到核心态，从而可以访问相应的资源。这样做的好处是：①使编程更加容易；②有利于系统安全，因为接口统一，所以可以对用户的参数进行检查；③接口统一有利于移植。

Linux 2.6 内核提供了近 300 个系统调用。

i386 支持两种不同方式的系统调用：

（1）执行“int $0x80 ”指令，在 2.6 之前的老版本一直采用这种方式。从核心态返回用户态的对应指令是 iret。

（2）执行“sysenter”指令，该指令在 Pentium Ⅱ时引入，执行速度比 int 要快。2.6 内核目前利用了该特性，根据芯片类型决定采用 int 方式还是 sysenter 方式。从核心态返回用户态的对应指令是 sysexit。

无论是哪种方式，它们都会执行下面一条语句：

```
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
```

寄存器 eax 中放的是系统调用号，sys_call_table 为系统调用表，该表保存的是各个系统调用的入口地址，下面列出了系统调用表的前三项：

```
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_ni_syscall)          /*第 0 号 */
```

```
.long SYMBOL_NAME(sys_exit)
.long SYMBOL_NAME(sys_fork)
```

可以看到，fork 系统调用号为 2，对应的系统调用实现函数为 sys_fork()，该函数又调用 do_fork()完成具体的工作，细节见第 1 章。许多库函数都有类似的形式，假设库函数名为 xyz，则其对应的系统调用名为 sys_xyz。sys_xyz()往往只做一些参数检查之类的工作，然后调用具体的函数进行处理。

6.4 中断的处理

与异常的处理相比，中断处理要复杂一些。中断处理分为两部分——上半部分和下半部分。本节先介绍上半部分的处理，然后再介绍下半部分的处理。

6.4.1 中断控制器

每个硬件设备的控制器都能通过中断请求线发出中断请求（简称 IRQ），所有设备的中断请求线又连到中断控制器的输入端。在 x86 单 CPU 的机器上采用两个 8259A 芯片作为中断控制器，一主一从。从 8259A 将它的 INT 引脚与主 8259A 的 IRQ2 相连，如图 6-1 所示。8259A 从 IRQ*n* 请求线上接收中断信号，每个请求线有一个触发器来保存请求信号，从而形成中断请求寄存器。当 8259A 有中断信号输入且中断信号不被屏蔽时，主 8259A 向 CPU 发出 INT 信号，请求中断。这时如果 CPU 是处于允许中断状况，CPU 就会发信号给 8259A 进入中断响应周期。

中断请求线的编号是从零开始的，在对 8259A 芯片的初始化过程中，第 *n* 号中断对应在 IDT 表中的索引为 *n*+32。很明显，因为中断请求线的数目有限，所以几个设备共享一根线是有必要的。当这根输入线有中断请求时，处理程序需要依次查询这些设备，以确定究竟是哪一个设备发出了中断请求。

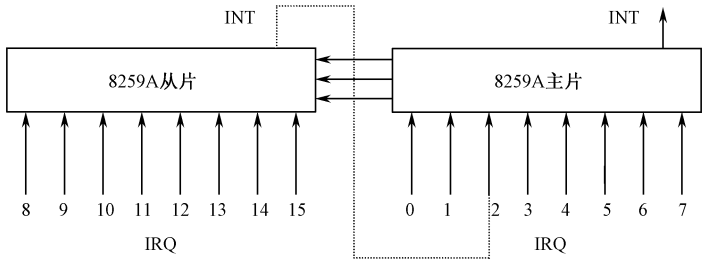

图 6-1 两个 8259A 芯片级联

6.4.2 管理中断的数据结构

核心的数据结构有两个：hw_interrupt_type 结构和 irq_desc_t 结构。

hw_interrupt_type 结构的定义如下：

```
struct hw_interrupt_type {
        const char * typename;
        unsigned int (*startup)(unsigned int irq);
```

```
        void (*shutdown)(unsigned int irq);
        void (*enable)(unsigned int irq);
        void (*disable)(unsigned int irq);
        void (*ack)(unsigned int irq);
        void (*end)(unsigned int irq);
        void (*set_affinity)(unsigned int irq, cpumask_t dest);
};
typedef struct hw_interrupt_type    hw_irq_controller;
```

该结构用来描述中断控制器，Linux 支持多种中断控制器，hw_interrupt_type 是根据它们的共性抽象出来的一致接口。对于 8259A 芯片来讲，它的具体成员如下：

```
static struct hw_interrupt_type i8259A_irq_type = {
        "XT-PIC",
        startup_8259A_irq,
        shutdown_8259A_irq,
        enable_8259A_irq,
        disable_8259A_irq,
        mask_and_ack_8259A,
        end_8259A_irq,
        NULL
};
```

irq_desc_t 结构用来描述中断源，数组 irq_desc[ ]描述所有的中断源。

```
typedef struct irq_desc {
        hw_irq_controller *handler; /*该中断所属的中断控制器*/
        void *handler_data;
        struct irqaction *action;       /*IRQ  响应链*/
        unsigned int status;            /*IRQ  状态*/
        unsigned int depth;             /*嵌套深度*/
        unsigned int irq_count;
        unsigned int irqs_unhandled;
        spinlock_t lock;
} ____cacheline_aligned irq_desc_t;
extern irq_desc_t irq_desc [NR_IRQS];
```

irqaction 结构记录当中断发生时具体的处理函数，其定义如下

```
struct irqaction {
    irqreturn_t (*handler)(int, void *, struct pt_regs *);      /*具体的处理函数*/
    unsigned long flags;
    cpumask_t mask;
    const char *name;
    void *dev_id;
    struct irqaction *next;                         /*利用该成员形成链表*/
```

```
    int irq;
    struct proc_dir_entry *dir;
};
```

irq_desc_t 的 action 成员实际是一个 irqaction 结构的链表，因为多个设备可能共享一个中断源，每个设备都必须提供一个 irqction 结构挂入 action 链表。

每个设备在使用前必须注册一个 IRQ 号，并提供相应的处理函数。该功能是通过函数 request_irq()实现的。下面是软驱驱动程序的代码：

```
request_irq(FLOPPY_IRQ, floppy_interrupt, SA_INTERRUPT|SA_SAMPLE_RANDOM,"floppy",
            NULL);
```

其中，FLOPPY_IRQ 是 IRQ 号，floppy_interrupt()是处理函数。如果该函数成功，irq_desc [FLOPPY_IRQ]的 action 成员将会指向一个新分配的 irqaction 结构，该结构的 handler 指向 floppy_interrupt()。

6.4.3 中断的处理过程

下面描述中断的处理流程：

① 中断信号由外部设备发送到中断控制器，中断控制器根据 IRQ 号转换成相应的中断向量号传给 CPU 。

② CPU 接收中断后，保存现场，根据中断向量号到 IDT 中查找相应的处理函数。这些处理函数的操作类似，都是先将中断向量号入栈，然后将硬件未自动保存的某些寄存器的内容入栈，再调用 do_IRQ()函数。

③ do_IRQ()函数完成一系列的动作，包括对中断控制器确认，设置中断源状态等，接着它会根据 IRQ 号找到描述中断具体动作的 irqaction 结构变量 action，执行如下代码：

```
do {
    ret = action->handler(irq, action->dev_id, regs);
    if (ret == IRQ_HANDLED)
        status |= action->flags;
    retval |= ret;
    action = action->next;
} while (action);
```

while 循环是用来处理设备共享中断号的情况。

④ 最后，do_IRQ()函数检查是否有软中断，如有，则调用 do_softirq()执行软中断。

⑤ 跳转到 ret_from_intr 退出，恢复中断前的现场。

6.5 软中断

前面已经提到，中断处理分为上半部分和下半部分，这样做是因为 CPU 在响应中断并进行处理的时候往往会关闭中断，而在此期间完全可能有多个设备发出中断请求，所以希望能够尽快地响应中断以免中断丢失。

下面以网卡驱动程序为例来描述解决的办法。当有网卡接收到数据包时，向 CPU 发出中

断后处理分成两部分：

① CPU 响应，关闭中断，把数据包从硬件缓冲区复制到内核空间，做标记用来激活下半部分的动作，开中断。

② 处理数据包。

注意到下半部分是可以在开中断的情况下运行，并且第二步并不一定紧接着第一步运行（如第一步后马上又有中断的情况），实际情况往往是有点时延。如此便保证了中断的响应速度，大部分驱动程序都遵循这种处理方法。习惯上把下半部分称为软中断，而上半部分称为硬中断。

软中断可以并发执行，相同的与不同的软中断都可以在不同的 CPU 上同时执行。进一步加强条件，就得到了一种特殊的软中断——tasklet，相同的 tasklet 不会同时在不同的 CPU 上运行，不同的 tasklet 可以在不同 CPU 上同时运行。

系统预定义了 6 个软中断：

HI_SOFTIRQ——用于处理高优先级别的 tasklet。

TIMER_SOFTIRQ——用于时钟中断。

SCSI_SOFTIRQ——用于 SCSI 总线中断。

TASKLET_SOFTIRQ——对应于 tasklet。

NET_TX_SOFTIRQ——用于发送网络数据。

NET_RX_SOFTIRQ——接收网络数据。

软中断定义要使用 open_softirq()函数，下面是注册 NET_RX_SOFTIRQ 软中断的语句：

```
open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

其中，NET_RX_SOFTIRQ 为软中断号。net_rx_action 是一个函数，当软中断被处理时调用该函数。raise_softirq()和 raise_softirq()两个用来设置标志位激活软中断，该函数在中断处理的上半部分被调用，例如下面的语句激活 NET_RX_SOFTIRQ 软中断：

```
raise_softirq_irqoff(NET_TX_SOFTIRQ);
```

内核的 do_softirq()函数用于处理软中断，它发现哪个软中断的标志位被设置，就调用相应的软中断处理函数。do_softirq()在三个地方得到执行机会。

（1）irq_exit()函数被调用时。该函数被调用的场合有：

① 当 do_IRQ()结束中断上半部分的处理时。

② 支持 APIC 的系统，执行本地时钟中断处理函数 smp_apic_timer_interrupt()结束时。

③ SMP 系统执行 smp_call_function_interrupt()函数结束时。

（2）当一个内核线程 ksoftirqd/n（n 代表 CPU 号）被调用时。

（3）当 local_bh_enable()函数被调用时。

在上述三个执行时机中，如果软中断标志位被激活，则调用 do_softirq()。

第 7 章　Sys V进程间通信

Linux 支持多种进程间的通信方式，如管道、信号、SysV 进程间通信和套接口等。本章将介绍 SysV 进程间通信的三种手段：信号量、消息队列和共享内存。

SysV 进程间通信的内核实现在接口和数据结构两方面与库函数有许多相似之处，在下面的内容中请注意两者之间的区别。

7.1　共有的特性

先介绍消息队列、信号量、共享内存在操作界面上的共同处。三者均有 xxxget()函数及 xxxctl()函数，xxx 代表 msg，sem，shm 之中的任何一个，这里只讨论函数共同拥有的参数或标志位，特定的则不涉及。

下面使用术语 IPC 资源来表示单独的消息队列、共享内存或信号量集合。xxxget()函数有两个共同的参数：key 和 oflag。key 既可由 ftok()函数产生，也可以是 IPC_PRIVATE 常量，key 值是 IPC 资源的外部表示。oflag 包括读/写权限，还可以包含 IPC_CREAT 和 IPC_EXCL 标志位。它们组合的效果如下：

① 指定 key 为 IPC_PRIVATE 保证创建一个唯一的 IPC 资源。

② 设置 oflag 参数的 IPC_CREATE 标志位，但不设置 IPC_EXCL。如果相应 key 的 IPC 资源不存在，则创建一个 IPC 资源，否则返回已存在的 IPC 资源。

③ oflag 参数的 IPC_CREATE 和 IPC_EXCL 位同时设置。如果相应 key 的 IPC 资源不存在，则创建一个 IPC 资源，否则返回一个错误信息。

xxxctl()均提供 IPC_SET，IPC_STAT，IPC_INFO 和 IPC_RMID 命令。前两者用来设置或得到 IPC 资源的状态信息，IPC_INFO 用来得到 IPC 资源信息，IPC_RMID 用来释放 IPC 资源。

信号量、消息队列和共享内存都是先通过 xxxget()创建一个 IPC 资源，返回值是该 IPC 资源 ID。在以后的操作中，均以 IPC 资源 ID 为参数对相应的 IPC 资源进行操作。注意 IPC 资源 ID 不是局限于进程的，而是全局可见的。只要权限允许，别的进程可以通过 xxxget()取得已有的 IPC 资源 ID 并对其操作，从而使进程间通信成为可能。

在内核中，每一类 IPC 资源都有一个 ipc_ids 结构的全局变量来描述同一类资源的公有数据，三个全局变量分别是 sem_ids，msg_ids 和 shm_ids。ipc_ids 结构定义如下：

```
struct ipc_ids {
    int in_use;                    /* entries 数组已使用的元素个数*/
    int max_id;
    unsigned short seq;
    unsigned short seq_max;
    struct semaphore sem;          /*内核信号量，控制对 ipc_ids 结构的访问*/
    struct ipc_id_ary nullentry;
    struct ipc_id_ary* entries;    /*变长数组，数组大小决定可用资源数*/
```

```
};
struct ipc_id_ary {
        int size;
        struct kern_ipc_perm *p[0];
};
```

ipc_id_ary 结构的成员 size 表示成员数组 p 的大小，也就是 entries 数组的大小，系统管理员可动态调整该数组的大小。数组 p 的每一项指向一个 kern_ipc_perm 结构。kern_ipc_perm 结构表示每一个 IPC 资源的属性，用来控制操作权限。其定义如下：

```
struct kern_ipc_perm {
        spinlock_t  lock;
        int         deleted;
        key_t       key;            /*键值，由用户提供，为 xxxget()所用
        uid_t       uid;            /*创建者用户 ID*/
        gid_t       gid;            /*创建者组 ID*/
        uid_t       cuid;           /*所有者用户 ID*/
        gid_t       cgid;           /*所有者组 ID*/
        mode_t      mode;           /*操作权限，包括读、写等*/
        unsigned long     seq;
        void        *security;
};
```

数组 p 的各项均初始化为 NULL，当创建一个 IPC 资源时，均会调用函数 ipc_addid()从相应 ipc_ids 结构的 entries 数组中找出第一个未使用的项，然后返回其下标 index。注意，index 并不是返回的 IPC 资源 ID，但它们之间存在如下关系：

$$\text{IPC 资源 ID} = \text{SEQ\_MULTIPLIER} \times \text{seq} + \text{index}$$

式中，SEQ_MULTIPLIER 是可用资源的最大数目，seq 是 ipc_ids 结构中的 seq。每当分配一个 IPC 资源时，ipc_ids 结构中的 seq 就增一。IPC 资源 ID 号是一个动态申请和释放的资源，不希望 ID 号被释放后又立即被分配给别的资源，因为这样可能会导致误用。通过变换 seq 值，短期内即使复用 index 也会得到不同的资源 ID 号。

当知道 IPC 资源 ID 时，对 SEQ_MULTIPLIER 取余就可得到其在 entries 数组中的 index，从而找到相应的 IPC 资源。

7.2 信号量

信号量是具有整数值的对象，它支持 P，V 原语。进程可以利用信号量实现同步和互斥。有必要指出，SysV 信号量是用户空间的操作，最终需要内核的同步机制支持。

SysV 支持的信号量实质上是一个信号量集合，由多个单独的信号量组成。在后面的叙述中，SysV 信号量称为信号量集合，而单个的信号量直接称为信号量。支持信号量集合的原因是，进程往往需要在同时具备多个资源的情况下才能工作，典型的例子是银行家算法。

信号量集合在内核中用结构 sem_array 表示，其定义如下：

```
struct sem_array {
        struct kern_ipc_perm   sem_perm;
        time_t                 sem_otime;          /*  最近一次操作时间  */
        time_t                 sem_ctime;          /*  最近一次的改变时间  */
        struct sem             *sem_base;
        struct sem_queue       *sem_pending;       /*  挂起操作队列*/
        struct sem_queue       **sem_pending_last;
        struct sem_undo        *undo;
        unsigned long          sem_nsems;
};
```

其中，sem_base 指向第一个信号量，sem_nsems 表示信号量的个数。信号量集合中的每一个信号用结构 sem 表示，其定义如下：

```
struct sem {
        int    semval;          /*  信号量的当前值  */
        int    sempid;          /*  最近对信号量操作进程的 pid */
};
```

信号量的初始值可以调用函数 semctl()进行设置。用户可以调用函数 semop()对信号量集合中的一个或多个信号量进行操作。semop()函数原型如下：

int　　semop(int semid, struct sembuf *opsptr, size_t nops);

其中，semid 为 IPC 资源 ID；opsptr 为操作的集合；nops 为数组 opsptr 的大小。

需强调的是，内核必须保证操作数组 opsptr 原子地执行。要么完成所有操作，要么什么也不做。每一个操作都是 sembuf 结构变量，该结构在函数库和内核中均有定义，其定义如下：

```
struct sembuf {
    unsigned short   sem_num;     /*  在 sem_array.sem_base[]数组中的下标*/
    short            sem_op;
    short            sem_flg;
};
```

（1）sem_numr 指明是对哪一个信号操作。

（2）sem_flag 指明一些操作标志位，可以有下述值：

① SEM_UNDO。当进程结束但还拥有信号量资源时，应将信号量资源返还给相应的信号量集合。内核有一个 sem_undo 结构用于跟踪这方面的情况，进程描述符有一个 semundo 成员记录进程在这方面的信息。在下面关于 semop 的讨论中省略了这方面的信息。

② IPC_NOWAIT。当操作不能立即完成时，若 IPC_NOWAIT 被设置，进程立即返回，否则进程进入阻塞状态等待时机成熟时被唤醒，以完成该操作。

（3）sem_op 指定具体的操作，它的值有如下含义：

① 若大于 0，则将该值加到信号量的当前值上。

② 若等于 0，则用户希望信号量的当前值变为 0。如果值已经是 0，则立即返回。如果不是 0，则取决于 IPC_NOWAIT 是否被设置。

③ 若小于 0，则要看信号量的当前值是否大于等于 sem_op 的绝对值。如果大于等于，就从信号量的当前值中减去 sem_op 的绝对值。如果小于，则取决于 IPC_NOWAIT 是否被设置。

当进程的信号量操作不能完成而进入阻塞状态时，需要将一个代表着当前进程的 sem_queue 结构链入相应的信号量集合的等待队列，即 sem_array 结构的 sem_pending 队列。sem_queue 结构结构定义如下：

```
struct sem_queue {
    struct sem_queue    * next;          /*队列中的下一个元素*/
    struct sem_queue    **prev;          /*队列中的前一个元素*/
    struct task_struct  *sleeper;        /*阻塞进程的描述符*/
    struct sem_undo     *undo;
    int    pid;                          /*阻塞进程的 pid */
    int    status;
    struct sem_array    *sma;            /*sem_queue 结构所属的信号量集合*/
    int    id;
    struct sembuf       *sops;           /*挂起的操作数组*/
    int    nsops;                        /*挂起的操作个数*/
    int    alter;
};
```

图 7-1 显示了上述数据结构之间的关系。

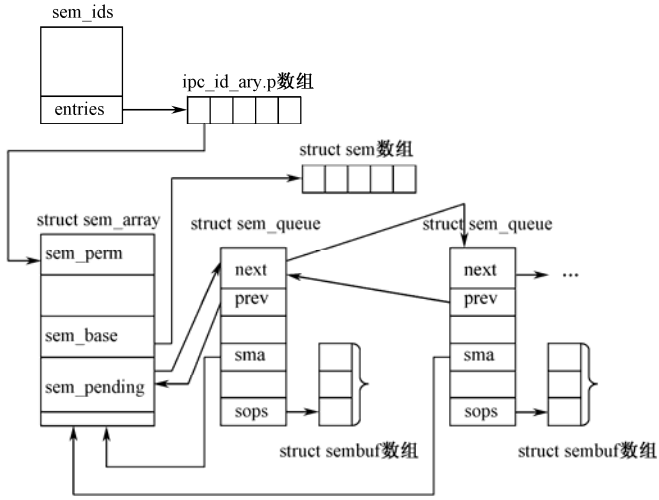

图 7-1　信号量各数据结构之间的关系

7.3 消息队列

具有写入权限的进程可以往消息队列中写入一个消息，具备读出权限的进程，则可以从消息队列中读出一个消息，这就是消息队列支持进程通信的方式。

除前面提到的两个函数外，用户还可以使用另外两个函数操作消息队列。msgsnd()函数

将消息放入队列中，其原型如下：

int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);

（1）msqid 为消息队列的资源 ID 号。

（2）msgp 为消息缓冲区的首地址。消息缓冲区由两部分组成，首先是消息的类型，然后才是数据部分。

（3）msgsz 为消息缓冲区的长度。

（4）msgflg 可以是 0，也可以是 IPC_NOWAIT。

另一个函数 msgrcv()从某个消息队列中读一个消息并将其移出消息队列。其原型如下：

int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);

（1）msgp 为接收消息的缓冲区首址。

（2）msgsz 为接收缓冲区的大小，这是函数能返回的最大数据量。

（3）msgtyp 指定接收消息的类型。分为如下三种情况：

① 值为 0，返回队列中的第一个消息。

② 值大于 0，返回类型为 msgtype 的第一个消息。

③ 值小于 0，则返回类型值小于或等于 msgtype 的绝对值的消息中类型值最小的第一个消息。

（4）msgflg 可以是 IPC_NOWAIT 或 MSG_NOERROR。MSG_NOERROR 允许消息长度大于接收缓冲区长度时截断消息返回。

在内核中消息队列用 msg_queue 结构表示，其定义如下：

```
struct msg_queue {
    struct kern_ipc_perm q_perm;
    time_t q_stime;                 /*最近一次 msgsnd 时间*/
    time_t q_rtime;                 /*最近一次 msgrcv  时间*/
    time_t q_ctime;                 /*最近的改变时间*/
    unsigned long q_cbytes;         /*队列中的字节数*/
    unsigned long q_qnum;           /*队列中的消息数目*/
    unsigned long q_qbytes;         /*队列中允许的最大字节数*/
    pid_t q_lspid;                  /*最近一次  msgsnd()发送进程的 pid */
    pid_t q_lrpid;                  /*最近一次  msgrcv()接收进程的 pid */

    struct list_head q_messages;    /*消息队列*/
    struct list_head q_receivers;   /*待接收消息的阻塞进程队列*/
    struct list_head q_senders;     /*待发送消息的阻塞进程队列*/
};
```

若 IPC_NOWAIT 未被设置，则当消息队列的容量已满时，发送消息的进程会进入阻塞状态并添加到相应的 q_senders 队列，而当消息队列中无合适的消息时，接收进程会进入阻塞状态并添加到相应的 q_receivers 队列。消息队列中的每个消息都链入 q_message 队列中，每个消息用一个 msg_msg 结构描述，该结构定义如下：

```
struct msg_msg {
        struct list_head m_list;     /*消息队列链表*/
        long    m_type;              /*消息的类型  */
        int m_ts;                    /*消息的长度  */
        struct msg_msgseg* next; /*链接属于这个消息的下一个消息片*/
        void *security;
};
struct msg_msgseg {
         struct msg_msgseg* next;
};
```

可以看出，msg_msg 结构实际只是一个消息头部，并不包含消息的数据部分。Linux 的处理方法如下：数据部分的空间紧接 msg_msg 结构，从而保证了可以找到属于该消息的数据，但是当数据部分的空间与 msg_msg 结构所占空间大于一个页面时，则将其以页面为单位分片。第一个页面存储 msg_msg 结构与首部分数据，随后的再分配空间则存储 struct msg_msgseg 结构与剩余的数据。如果这两者所占空间之和仍大于一个页面，则继续分配下去。msg_msgseg 结构用以把消息片链接在一起。图 7-2 给出了各个数据结构之间的联系。

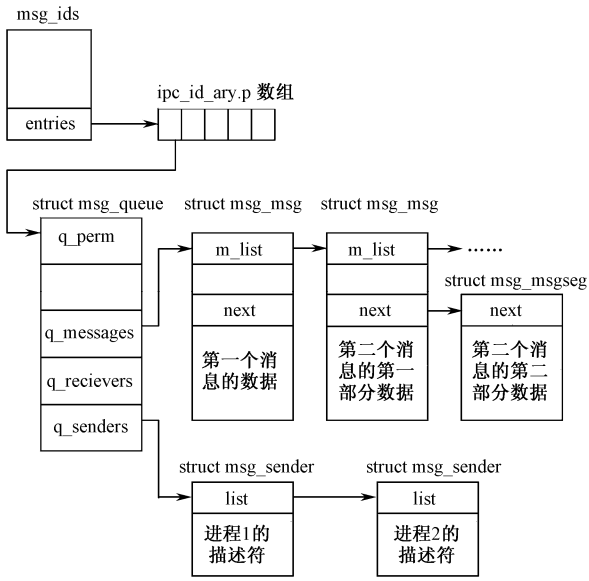

图 7-2　消息队列各数据结构之间的关系

7.4 共享内存

共享内存是多个进程共享的一块内存区域。不同的进程可把共享内存映射到自己的一块地址空间，不同的进程进行映射的空间地址不一定相同。映射后，进程对该段地址空间的写操作就是对共享内存写操作。同样地，映射了该共享内存的其他进程，就可以看见该变化，从而实现进程通信。但是共享内存没有提供进程同步与互斥的机制，所以往往需要和信号量

配合使用。

与其他进程通信方式相比，共享内存在进行数据交换方面效率比较高。如图 7-3 所示，假设进程 A 给进程 B 发送数据，我们来比较一下消息队列与共享内存的传送效率。如果采用消息队列，A 进程调用 msgsnd()需要从用户态切换到核心态，然后再返回用户态，B 进程调用 msgrcv()需要从用户态切换到核心态，然后再返回用户态，总共需要 4 次切换。如果采用共享内存方式，假设映射已经建立，A 往共享内存写数据，B 立即能看见，根本无须运行状态的切换。

shmget()函数有一个参数指定共享内存区域的大小，该函数建立的共享内存区在内核中用 shmid_kernel 结构表示，其定义如下：

```
struct shmid_kernel
{
        struct kern_ipc_perm   shm_perm;
        struct file *          shm_file;
        int                    id;
        unsigned long          shm_nattch;     /*已建立映射的数目*/
        unsigned long          shm_segsz;      /*共享内存区的大小*/
        time_t                 shm_atim;
        time_t                 shm_dtim;
        time_t                 shm_ctim;
        pid_t                  shm_cprid;
        pid_t                  shm_lprid;
        struct user_struct *mlock_user;
};
```

shmget()创建的共享内存区域并没有立即分配物理内存，而是创建一个文件对象 shm_file 来描述该区域，而该文件属于 shm 文件系统。shm 文件系统是一个内存文件系统，它不依赖于磁盘文件的内容，记载该文件系统的相关信息随着关机彻底消失。在后面的介绍中可以看到，shm_file 实际上是由一个个页面组成的。

进程调用 shmat()函数建立进程地址空间与共享内存区的映射。选取进程地址空间的哪一段区间进行映射，可由用户指定，也可委托内核选择。shmat 函数找到区间后进程分配一个 vm_area_struct 结构描述该区间，vm_area_struct 结构的各项被初始化，其中 file 成员被初始化为 shm_file，而 vm_ops 成员被初始化为 shm_vm_ops，变量 shm_vm_ops 定义如下：

```
static struct vm_operations_struct shm_vm_ops = {
        open: shm_open,
        close:shm_close,
        nopage:     shmem_nopage,
        ……
};
```

需注意的是，共享内存区在第一次 shmat()后仍然没有分配物理内存，Linux 在此采取的仍然是“懒惰策略”。当进程第一次访问该映射共享内存区的区间地址时，将触发页面异常，

根据第 3 章中有关页面异常处理的内容，我们知道最终将调用 shmem_nopage()函数。该函数处理的过程如下：

① 先根据文件和文件位置查找页面是否已经在 page cache 中，因为别的进程可能已经为映射的共享内存区页面申请了一个物理页帧。如果找到，修改本进程页表即可。否则继续下一步。

② 检查被映射的共享内存区页面是否被访问过，并已被换出到交换分区。如果是，则调入该页面，修改进程页表。否则继续下一步。

③ 若被映射的共享内存区页面从未被访问过，则向内存子系统申请一个物理页帧，修改进程页表。

进程可以调用 shmdt()函数解除地址空间与共享内存区的映射关系，其处理较为简单，主要是修改页表及释放 vm_area_struct 结构。图 7-3 显示出各部分之间的联系。

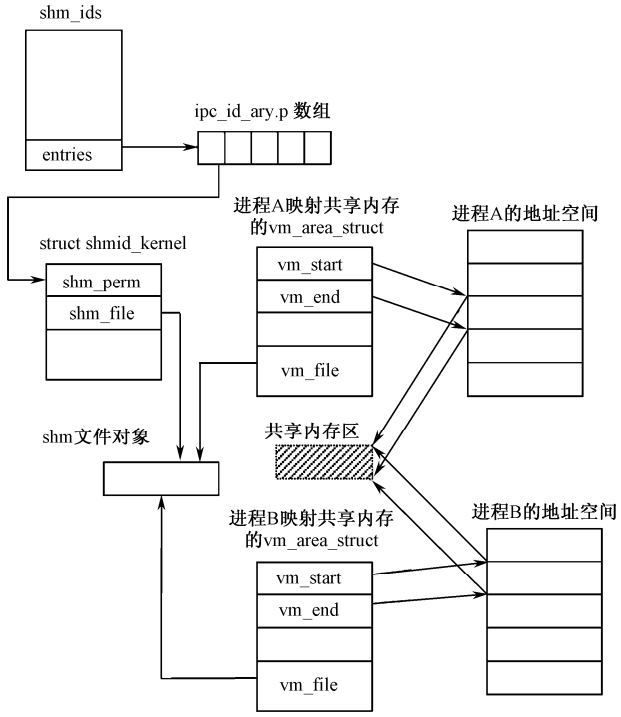

图 7-3　共享内存各数据结构之间的关系

第二部分

基于Linux操作系统的实验

Part two

本书的实验分成两大类：用户态编程实验和内核编程实验。内核编程实验又分为基础实验和综合实验两个子类。在设计实验的过程中，充分考虑了实验内容由易到难、由浅入深。对于 Linux 初学者而言，较好的学习方式是先从外围入手，熟悉操作系统提供给用户的接口，通过用户态编程观察、理解操作系统的行为和原理，然后在此基础上更进一步深入透彻地理解。应当看到，透彻地了解操作系统的最佳方式莫过于投身内核编程实践，但当前 Linux 内核的代码已达七百万行，容易使人产生畏难情绪。这里安排的内核编程基础实验能训练用户内核编程的基本技能，内核编程综合实验能更全面地提升用户的内核编程能力。

用户态编程实验由三个实验组成，这三个实验从不同的角度强调了操作系统与用户的接口：第一个接口是用户日常操作使用的 shell，第二个接口是系统调用，第三个接口是 proc 文件系统。实验 1 是 shell 脚本编程，脚本批量打包命令，方便系统的日常维护管理，对于进行本书的某些后续实验来说，写一个脚本测试实验程序能极大减轻测试的工作量。实验 2 揭示了 shell 的运行原理，要求用户能利用内核提供的系统调用进行编程。实验 3 则从 proc 文件系统的角度，让用户观察系统的行为。该组实验可以作为操作系统原理课程的课后实验。

内核编程有两个基本功，一是内核模块编程，二是内核编译，前者是实现内核新功能及驱动的重要机制，后者是生成内核映像文件必

不可少的过程。在内核基本实验的具体实验内容设置上，我们首先考虑了从用户态到核心态的递进，从使用系统调用到实现系统调用，从用户态观察 proc 文件到在内核态创建 proc 文件。这部分实验应该由学生根据综合实验的需要自行实验。

内核编程综合实验的第一个实验，即实验 7，是进程隐藏，全面综合了基础实验的各个知识点，并且加大了难度。在基础实验中，内核态创建 proc 文件只需要使用内核相关函数，而在进程隐藏实验中，必须修改 proc 文件系统本身的实现过程，这也体现了难度渐进的原则。驱动程序不仅在内核代码中分量最大，而且市场对 Linux 驱动开发人员的需求一向不少，为此我们安排了两个实验，一个是关于字符设备的，另一个是关于 USB 盘驱动的。文件系统在操作系统中占据非常重要的位置，前面多个实验都要求对文件系统的内部实现有一定的了解，为此我们设计了 naivefs 文件系统实验，把一个较为复杂的文件系统的开发过程分解成多个步骤，以增量方式进行开发，每个步骤都可以测试，做完该实验的用户必定对系统调用层面到 VFS 抽象层再到具体的物理文件层有更加深入的了解。该部分实验只要求选做，或作为课程设计题目。

对于初次接触内核编程的读者来说，在开发中遇到问题时，建议多阅读内核源代码的相关部分，往往能发现值得借鉴甚至可以借用的代码。另外，通过网络能找到大量有用的信息，尤其是内核邮件列表含有很多设计的演变过程，这种过程是只看代码无法了解到的。

第 8 章　用户态编程实验

8.1　实验 1——bash脚本编程

bash 是 Linux 环境下常用的一个命令解释器，它既可以接受来自命令行的输入，也可以接受来自脚本文件的输入。脚本广泛用于日常的管理、维护及程序测试。本实验要求读者学习 bash 脚本编程的基本知识后，编写两个脚本小程序，为进一步提高脚本编程能力和后续实验的测试工作打好基础。

8.1.1　实验内容

（1）写一个脚本文件 checkuser，该脚本运行时带一个用户名作为参数。具体要求如下：

① 如果命令行格式不符合要求，应有错误提示信息。

② 在/etc/passwd 文件中查找是否有该用户，如有，则输出"Found <user> in the /etc/passwd file."；否则，输出"No such user on our system."。

（2）写一个脚本文件 printnumber，该脚本运行时带一个数值参数，参数可包含小数部分。具体要求如下：

如果命令行格式不符合要求，应有错误提示信息。

小数点前从个位数起每三位作为一节，节与节之间应该有逗号分开，如下所示：

$ printnumber 1234625

1,234,625

$printnumber 123462532.433

123,462,532.433

8.1.2　bash脚本编程简介

8.1.2.1　注释和简单命令

我们先看一个简单的 bash 脚本文件 hello：

```
#!/bin/bash
# name: hello
echo 'hello, world!'
```

如果要运行该脚本程序，首先要给它添加可执行权限：

$chmod +x hello

$./hello

脚本中可以加入注释，注释以字符#开始，至行尾结束。例如，文件hello的第二行就以

注释的形式说明了该脚本的文件名。脚本的第一行有点特殊，它以＃!/bin/bash开始[①]，表示这是一个bash脚本，应该用命令解释器bash来解释执行。

文件 hello 的第三行是一条简单命令 echo，它输出一个字符串'hello, world!'。bash 脚本的简单命令分为两种：内部命令和外部命令。前者由 bash 直接执行，如 cd，export，pwd，alias，cho，read 和 source 等；后者需要调用相应的可执行文件，如 ls，rm，vi，less，mount，telnet，tar，make 和 gcc 等。各命令的具体功能可参见附录 A 和相应的联机帮助。

另外，需要对撇号作特殊说明。在脚本 hello 的第三行，字符串'hello, world!'用单撇号括起来，意思是说不对其中的特殊字符（这里是字符“!”）进行扩展。字符“!”是 bash 的历史扩展字符，具有特殊的含义。关于各种撇号的使用方法，可以参考 bash 的联机帮助。

8.1.2.2 环境变量

正如 C 语言中可以定义数据变量一样，在 bash 脚本中也可以定义环境变量，方法如下：

```
msg='hello, world!'
```

以上命令定义了一个环境变量 msg，它包含字符串'hello, world!'。注意等号＝的两边不能有空格。通过在环境变量前加一个字符$，可以使用该环境变量的值：

```
echo $msg
```

这种方式在 bash 中称为“变量扩展”。为了将环境变量与周围的文字分开，也可以使用${msg}这种形式。

环境变量有一个特别好的用处：可以向子进程传递数据，而无论该子进程是一个普通的可执行文件还是另一个脚本，具体方法为

```
export  $msg
```

这样在以后启动子进程时，变量 msg 就传过去了，可以在该子进程中使用此变量。

另外，还有一些只读的特殊环境变量。如$1，$2，$3，… 分别表示调用此脚本时的第一、二、三……个参数，$0 表示带全路径的脚本名，$#表示脚本的参数个数，$*表示由所有参数构成的一个单字符串，$?表示最近运行的命令的退出状态。

8.1.2.3 控制结构

类似于 C 语言，bash 脚本语言也有控制机制，如条件结构有 if 和 case 语句，循环结构有 for，while 和 until 语句。**请注意**：在下面的语法描述中，分号也可以用一个或多个换行符替换。

1．if语句

if 语句的语法为

```
if test-commands; then
        consequent-commands;
[elif more-test-commands; then
        more-consequents;]
```

① 如果可执行文件 bash 安装在其他目录下，那么脚本第一行中的/bin/bash 应改为实际的全路径名。

[else alternate-consequents;]

fi

其含义是：如果命令列表 test-commands 的返回值为 0（0 表示命令成功执行，相当于 C 语言的真；其他返回值表示假），则下一步执行命令列表 consequent-commands，否则执行后续的 elif 语句中的命令列表，如果它返回 0，则执行相应的 more-consequents 命令列表。elif 语句可以有多个，也可以没有。如果最后一个 if 或 elif 后面的命令列表返回非 0 值，那么就执行命令列表 alternate-consequents。最后执行的命令的返回值为整个 if 语句的返回值。

下面以脚本 test-if 为例进行说明：

```
#!/bin/bash
# name: test-if
if [ "${1##*.tar.}" = "gz" ]
then
        echo It appears to be a tarball zipped by gzip.
else
        echo It appears to NOT be a tarball zipped by gzip.
fi
```

此脚本判断一个文件名是否像一个被 gzip 压缩的 tar 文件的名字。值得说明的是，第三条语句中[ ]是 bash 的一个内部命令，表示对其中的条件表达式进行测试；"${1##*.tar.}" = "gz" 是一个条件表达式，表示判断"${1##*.tar.}"是否等于字符串"gz"；"${1##*.tar.}"表示对环境变量$1（脚本的第一个命令行参数）作截断处理之后的字符串，具体过程解释如下。从左至右扫描字符串$1，从$1 的第一个字符开始发现满足模式*.tar.的最大子串，然后将其截取，剩下的字符串即为结果。

在条件表达式中，除了用来判断两个字符串是否相等的运算符=外，还有如下其他一些比较运算符。

文件比较运算符有：

-e *filename*: 判断文件 *filename* 是否存在

-d *filename*: 判断文件 *filename* 是否为目录

-f *filename*: 判断文件 *filename* 是否存在且为常规文件

-r *filename*: 判断文件 *filename* 是否可读

-w *filename*: 判断文件 *filename* 是否可写

-x *filename*: 判断文件 *filename* 是否可执行

字符串比较运算符有：

-z *string*: 判断字符串 *string* 的长度是否为零

-n *string*: 判断字符串 *string* 的长度是否非零

string1 != *string2*: 判断字符串 *string1* 与 *string2* 是否不等

算术比较运算符有：

num1 –eq *num2*: 判断整数 *num1* 与 *num2* 是否相等

num1 -ne *num2*: 判断整数 *num1* 与 *num2* 是否不等

num1 -lt *num2*: 判断整数 *num1* 是否小于 *num2*

num1 -gt *num2*: 判断整数 *num1* 是否大于 *num2*

关于比较运算符的完整列表，可见参考文献[7]。

2．case语句

case 语句的语法为

```
case word in
[ [(] pattern [| pattern]...)
        command-list ;;]
...
esac
```

其含义是：寻找与 word 相匹配的第一个 pattern，然后执行对应的命令列表 command-list；如果不匹配任何模式或字符串，则不执行任何代码行。

下面以脚本 test-case 为例说明：

```
#!/bin/bash
# name: test-case
if [ -e "$1" ]
then
   case  ${1##*.tar.}  in
         bz2)
               tar  jxvf  $1
               ;;
         gz)
               tar  zxvf  $1
               ;;
         *)
               echo "wrong file type"
               ;;
   esac
else
   echo "the file $1 does not exist"
fi
```

此脚本首先判断第一个命令行参数所指的文件是否存在，若存在则再判断它是否为一个后缀为.tar.gz 或.tar.bz2 的文件，并分别作解压缩处理。在此脚本中，如果文件存在，则至少要执行一个代码块，因为任何不与 "bz2" 或 "gz" 匹配的字符串都将与 "*" 模式匹配。

3．for语句

for 语句的语法为

```
for  name  [in  words ...];  do  commands;  done
```

其含义是：对于 words 中的每一个字符串，都将其赋给变量 name，然后执行命令列表 commands。举例说明如下：

```
#!/bin/bash
# name: test-for
for  para  in  "$@"
do
        echo  ${para}
done
```

执行命令行：

$./test-for a b c

运行结果：

a

b

c

这个脚本遍历每一个命令行参数，并将其显示出来。$@是一个特殊的环境变量，前面已经提到过，表示由所有命令行参数构成的单字符串。

4．while和until语句

while 语句的语法为

while test-commands; do consequent-commands; done

其含义是：只要命令列表 test-commands 的返回结果为 0，就执行命令列表 consequent -commands，不断重复此过程，直至 test-commands 的返回结果为非 0。举例说明如下：

```
#!/bin/bash
unset  var
while [  "$var" != "end" ]
do
        echo  -n  "please input a string(\"end\" to exit): "
        read  var
        if [ "$var"  =  "end" ]
        then
            break
        fi
        echo  "the string you input is $var"
done
```

此脚本首先提示用户输入一个字符串，如果输入的是“end”，就结束；否则显示该字符串并继续读入下一个字符串。其中，“unset var”的功能是取消变量 var 的定义，“read var”的功能是读入一个字符串并将其存到变量 var 中。unset 和 read 都是 bash 的内部命令。

until 语句的语法与 while 语句类似，只是将关键字 while 替换为 until 即可。所不同的是其语义：until 语句退出循环的条件是命令列表 test-commands 的返回结果为 0，即正好与 while 语句相反。

8.1.2.4 函数

在 bash 脚本中，也可以定义与过程式语言（如 Pascal 和 C）类似的函数，并且 bash 脚本中的函数也可以接受和处理命令行参数，其方式类似于脚本对命令行参数的使用。举例说明如下：

```
#!/bin/bash
# name: test-function
myunzip()
{
        if [   -e   "$1" ]
        then
                case   ${1##*.tar.}   in
                        bz2)
                                tar   jxvf   $1
                                ;;
                        gz)
                                tar   zxvf   $1
                                ;;
                        *)
                                echo   "wrong file type"
                                ;;
                esac
        else
                echo   "the file $1 does not exist"
        fi
}
myunzip a.tar.gz
myunzip b.tar.gz
```

此脚本与前面介绍的 test-case 非常类似，只不过它采用了函数形式。从第三行到倒数第三行定义了一个函数 myunzip，在最后两行调用了此函数，并分别传递了一个命令行参数，该参数在函数 myunzip 中可通过$1 来访问。

需要说明的是，在 bash 脚本中，函数内部创建的环境变量是全局的，这意味着该变量在函数退出之后继续存在。如果希望在函数内部定义一个局部的环境变量，可以使用关键字 local。

8.1.3 实验指南

完成此次任务要用到 cut 命令，cut 可以从文件行中选择相应的部分输出到标准输出设备。其格式如下：

cut   [option] [File ...]

在此我们仅介绍要用到的 cut 命令选项：

① -d 字符。选项-d 指定的字符（称为定界符）把一行文字分割成多个段（field），如果未指定定界符，默认是制表符 Tab。

② -f List。选中段的列表。如果该行中没有定界符，则输出该行，除非指定了-s。

以下面的命令行为例：

$cut -d: -f1,3 /etc/passwd

该命令以冒号作为定界符，输出/etc/passwd 各行的第一和第三段。

8.2 实验 2——观察Linux行为

在掌握 Linux 操作系统基本概念的基础上，本实验将通过 proc 文件系统观察整个系统的一些重要特征，并要求编写一个程序，利用 proc 文件系统获得和修改系统的各种配置参数。本实验需要读者具备基本的 Linux 操作技能，以及基本的 C 语言编程能力。

8.2.1 实验内容

以超级用户的身份登录 Linux 系统，并进入/proc 目录，输入 ls 命令，查看该目录下的内容，同时查看每个文件的读、写权限。

（1）请回答下列问题：

① CPU 的类型和型号。

② 所使用的 Linux 版本。

③ 从启动到当前时刻经过的时间。

④ 当前内存状态。

（2）编写一个程序，用来获得内核参数（任意的参数均可）。

（3）编写一个程序，用来修改内核参数（任意的参数均可）。

8.2.2 proc文件系统简介

众所周知，我们在使用Windows的时候，只要按下【Ctrl+Alt+Delete】三个键，就能够看到如 图 8-1所示的“Windows任务管理器”，里面显示了当前的一些系统信息，如进程列表、内存使用情况等，为我们随时掌握系统状态提供了方便。

在 Linux 操作系统中，也提供了一套在用户态检查内核状态和系统特征的机制，即进程文件系统（process file system，简称 procfs）。比起 Windows 的任务管理器，proc 文件系统的功能更强大，它能提供更多的系统信息，甚至能修改部分系统信息，还能通过编写程序扩充其中的内容（见实验 5）。

早期的 UNIX 在设备文件目录/dev 下设置了一个特殊的文件/dev/mem，这个文件在现在的 Linux 系统中仍然存在，通过这个文件可以读/写系统的整个物理内存，而物理内存的地址就用作读/写时文件内部的位移量。这个特殊文件可以像普通文件一样，进行 read，write，lseek 等常规的文件操作，从而提供了一个在内核外部动态读/写包括内核映像和内核中各个数据结构及堆栈内容的手段。既可用于收集状态信息和统计信息，也可用于程序调试，还可以动态地改变一些数据结构或变量的内容。采用虚存以后，UNIX 又增加了一个特殊文件/dev/kmem，对应于系统的整个虚存空间。这两个特殊文件的作用和表现出来的重要性促使人们对其功能加以进一步的拓展，在系统中增设了一个/proc 目录，每当创建一个进程时就以其 pid 为文件名在这个目录下建立一个特殊文件，使得通过这个文件就可以读/写相应进程的用户空间和其他相关信息。经过多年的发展，procfs 成为了一个特殊的文件系统，这个文件系统所涵盖的

内容已几乎涉及系统的所有方面。

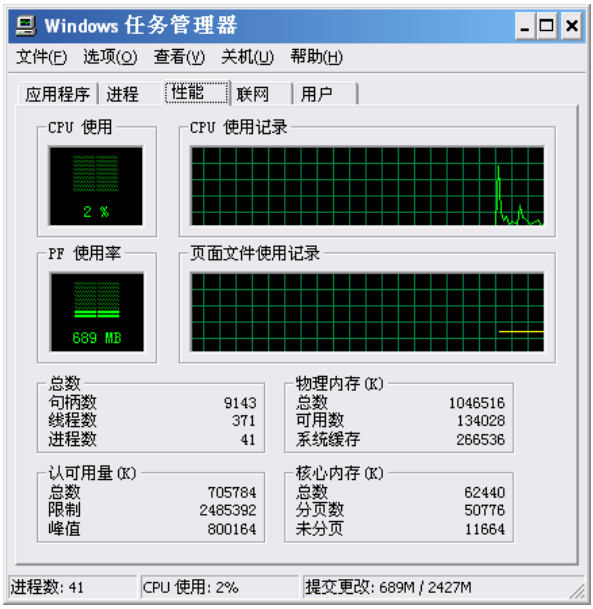

图 8-1　Windows 任务管理器

proc 文件系统将进程信息、系统的硬件信息（包括 CPU、内存状态及网卡等各种硬件设备）、系统相关机制（中断、I/O）等内容全部映射为虚拟的 Linux 文件。它以一种特殊的文件系统的方式，为访问系统内核数据的操作提供接口。也就是说，这个文件系统中所有的文件都是特殊文件，这些特殊文件一般与外部设备无关，所涉及的介质通常就是内存或 CPU 本身。当从一个特殊文件“读”出时，所读出的数据都是由系统内部按一定的规则临时生成的，或者从内存中收集、加工出来的，反之亦然。换言之，这些文件的内容都不存在于任何存储设备上，而是在读/写的时候才根据系统中的有关信息生成出来，或者映射到系统中的有关变量或数据结构。

下面来看 Linux 系统中 proc 文件系统的具体内容。首先，系统中当前运行的每一个进程都有一个对应的目录在/proc 下，以进程的 ID 号为目录名，它们就是读取进程信息的接口。进程目录中包含的内容如表 8-1 所示。

表 8-1

子目录/文件名称	内 容 描 述
Attr	进程属性
Auxv	ELF 解释器信息
Cmdline	进程的命令行
Cwd	进程的当前目录
Environ	进程环境变量的值
Exe	进程的可执行文件的链接
Fd	目录，包含所有打开的文件描述符
Loginuid	登录 UID
Maps	进程地址空间的各区间信息

续表

子目录/文件名称	内 容 描 述
Mem	进程的内存使用情况
Mounts	已安装文件系统信息
oom_adj	调节 oom-killer score 分值
oom_score	当前 oom-killer score 分值
Root	进程的根目录
Smap	进程地址空间每区间存储器使用情况
Stat	进程状态
Statm	进程存储器状态信息
Status	进程状态，以可读方式显示
task	目录，包含进程的子线程
wchan	进程等待通道

除去进程相关子目录，/proc 目录还包含一些目录/文件用于提供内核信息或者控制内核行为，表 8-2 中列出了一部分。

表 8-2

目 录 名	内 容 描 述
apm	高级电源管理信息
cmdline	内核命令行
cpuinfo	CPU 信息
devices	可用的设备信息
dma	已经使用的 DMA 通道
filesystems	系统支持的文件系统
interrupts	中断信息
ioports	端口使用信息
kallsyms	内核符号表
kcore	内核映像
kmsg	内核消息
loadavg	平均负载
locks	内核锁
meminfo	内存信息
misc	其他信息
modules	内核加载模块列表
mounts	已加载文件系统
partitions	系统识别的分区表
slabinfo	slab 对象池的信息
stat	全面信息统计状态表
swaps	交换分区使用情况
sys	内核参数
version	内核版本
uptime	系统正常运行时间

当然，读者的系统中未必有上述所有的目录和文件，这取决于内核的配置和装载的模块。

例如，/proc 目录中有两个很重要的子目录 net 和 scsi，而这两个子目录的存在均依赖于内核的配置。如果用户的系统不支持 SCSI 设备，则 scsi 目录不存在。

在/proc 目录中，大部分文件都属于 root，并且所有用户对其只拥有读权限，但有一个例外子目录 sys。sys 目录下的文件记录了内核各方面的运行参数，用户可以更改这些文件的值，结果是直接修改内核中的相应参数。下面举几个例子。

（1）/proc/sys/kernel/acct

该文件有三个可配置值，根据日志所在的文件系统上可用空间的数量（以百分比表示），这些值控制何时开始进行进程记账：

① 第 1 个值表示如果可用空间高于这个百分比值，则开始进程记账。

② 第 2 个值表示如果可用空间低于这个百分比值，则停止进程记账。

③ 第 3 个值检查上面两个值的频率（以秒为单位）。

要更改这个文件的某个值，应该回送用空格分隔的一串数字。默认设置为 4 2 30，也就是说，如果包含日志的文件系统上只有少于 2% 的可用空间，则这些值会使记账停止，如果有 4% 或更多可用空间，则再次启动记账。每 30 秒做一次检查。

（2）/proc/sys/fs/file-max

该文件指定了可以分配的文件句柄的最大数目。如果用户得到错误消息，说明由于打开文件数已经达到了最大值，而不能打开更多文件，则可能需要增加该值。

（3）/proc/sys/kernel/domainname

该文件允许配置网络域名。它没有默认值，可能已经设置了域名，也可能没有。

（4）/proc/sys/kernel/hostname

该文件允许配置网络主机名。同样，它也没有默认值，可能设置了主机名，或者没有。

（5）/proc/sys/kernel/printk

该文件有 4 个数字值，它们根据日志记录消息的重要性，定义将其发送到何处。关于不同日志级别的更多信息，请阅读 syslog(2) 联机帮助页。该文件的 4 个值依次为：

① 控制台日志级别。优先级高于该值的消息将被打印至控制台。

② 默认的消息日志级别。将用该优先级来打印没有优先级的消息。

③ 最低的控制台日志级别。控制台日志级别可被设置的最小值（最高优先级）。

④ 默认的控制台日志级别。控制台日志级别的默认值，默认设置为 7 4 1 7。

（6）/proc/sys/kernel/shmall

该文件是在任何给定时刻系统可以使用的共享内存的总量（以页为单位）。

（7）/proc/sys/kernel/shmmax

该文件指定内核所允许的最大共享内存段的大小（以字节为单位）。默认设置为 33554432。

（8）/proc/sys/kernel/shmmni

该文件表示用于整个系统共享内存段的最大数目，默认值为 4096。

如上所述，/proc/sys 目录不仅提供了内核信息，而且可以通过它修改内核参数。但是必须很小心，因为修改可能造成系统崩溃。要改变内核的参数，只要用 vi 编辑或用 echo 参数重定向到文件中即可。

8.2.3 实验指南

8.2.3.1 Linux环境下C语言编程环境简介

Linux 各发行版本按默认配置安装好以后，所需要的编程工具 gcc 一般均已安装，如若不然，则需要手动安装。

1．使用编辑器编写程序

用户喜欢使用命令行可以使用 vi 或 Emacs，喜欢使用图形界面编辑器的话，Gnome 下有 gedit，KDE 下有 kedit。在命令行键入如下字符即可：

$gedit or **$kedit**

编辑完成后保存文件，假定文件名为 hello.c。

2．编译生成可执行文件。

键入命令行：

$gcc –Wall –o hello hello.c

上面命令行利用编译器 gcc 编译 C 程序 hello.c，选项“-Wall”告诉编译器针对程序中可能存在问题的地方产生尽可能多的警告信息，“-o”选项用来指定输出文件名，“–o hello”表示生成可执行文件 hello。如果编译不出现错误，则可继续下一步。

另外一种生成可执行文件的方式是，用户先写一个 Makefile 文件，然后用户直接输入命令 make 就可以了。限于篇幅，我们在此不介绍如何撰写 Makefile，感兴趣的读者可以查阅相关资料。在后面多个实验中我们都是使用 Makefile 文件来生成目标模块。

3．执行程序

$./hello

注意要有./。

如果执行结果与预想不符，可使用调试器 gdb 找出错误，gdb 的用法请自行查阅相关资料。

8.2.3.2 实验程序框架

本实验完全可以不涉及内核编程，而只使用几个库函数。事实上，只需要编写一个简单的读文本文件的程序，就可以直接用于读 proc 文件系统中的文件。下面给出一个简单的程序框架，读者可以在此基础上添加自己的代码，从而完成上述实验。

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>
...
int main (int    argc, char* argv[])
{
        ...
        int fd=open(argv[1],FLAG,MODE);      //文件名作为参数传入。FLAG 和 MODE
```

```
                                                    //的值由打开的功能决定
        if(fd!=-1)
        {
        //读/写相应的内核参数
              ...
              close(fd);
        }
        else
        {
        //作错误处理
        }
        ...
        return EXIT_SUCCESS;
}
```

8.3 实验 3——实现Linux命令解释器

在 8.1 节实验 1 中我们已经知道，命令解释器可以解释执行用户的指令，实现用户与计算机之间的交互。在此，我们先给出一个简单的命令解释器 myshell，然后要求在此基础上添加相应的功能。

8.3.1 实验内容

本次实验的内容由下面几部分组成：

（1）分析且运行 myshell。

（2）扩充 myshell 功能，使其支持以下内部命令：

① cd <目录>——更改当前的工作目录到另一个<目录>。如果<目录>未指定，输出当前工作目录。如果<目录>不存在，应当有错误信息提示。

② echo <内容>——显示 echo 后的内容且换行。

③ help——简短概要地输出 myshell 的使用方法和基本功能。

④ jobs——输出 myshell 当前的一系列子进程，必须提供子进程的命名和 PID 号。

（3）添加重定向和管道功能。

8.3.2 myshell的语法

此命令解释器只能接受简单命令。命令行上的每一行输入都被视为一个简单命令，它由多个以空白字符（空格或制表符）分隔的词组成，其中第一个词是命令名，后面各词为命令参数，词（word）定义为不含空白字符和换行符的字符串。用 BNF 格式表示为

```
<simple_command> ::=   <word>
                     |  <simple_command> <word>
```

8.3.3 myshell的程序框架

程序的主执行框架为

for (;;){

1 显示提示符

2 读入一行命令

3 判断此命令是否为“exit”，若是则退出

4 分析并执行这行命令

}

主程序 myshell.c 如下所示：

```
#include   <stdio.h>
#include   <string.h>
#include   <limits.h>
#include   <unistd.h>
#include   <sys/types.h>

#define   PROMPT_STRING "[myshell]$ "
#define   QUIT_STRING "exit\n"

static   char   inbuf[MAX_CANON];
char*   g_ptr;
char*   g_lim;

extern   void   yylex();

int   main (void) {
        for( ; ; ) {
                if (fputs(PROMPT_STRING, stdout) == EOF)
                continue;
                if (fgets(inbuf, MAX_CANON, stdin) == NULL)
                        continue;
                if (strcmp(inbuf, QUIT_STRING) == 0)
                        break;
                g_ptr = inbuf;
                g_lim = inbuf + strlen(inbuf);
                yylex();
        }
        return 0;
}
```

需要特别说明的是，框架中的“4 分析并执行这行命令”，包含 myshell.c 中的三行语句：

g_ptr = inbuf;

g_lim = inbuf + strlen(inbuf);

yylex();

前面两行将命令行字符串的首、尾指针放在了全局变量 g_ptr 和 g_lim 中，具体分析执行该行命令的任务由第三行的函数 yylex()来完成，这是由语法分析工具 flex 生成的。

8.3.4 myshell命令行的语法分析

我们采用专业的语法分析工具 flex，此工具使用非常方便，也易于对语法进行扩展和修改。虽然 flex 包含许多内容，但这里我们只涉及其中很少的部分，读者通过简单的学习即可掌握。

8.3.3 节用到的 yylex()函数由 flex 根据一个输入文件 parse.lex 生成。此文件由三部分组成，下面分别进行介绍。

parse.lex 文件的第一部分如下，是 C 代码，包括后面要用的头文件、数据变量和函数原型。

```
%{
#include   <errno.h>
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <string.h>
#include   <sys/types.h>
#include   <sys/wait.h>

// input related
extern   char*   g_ptr;
extern   char*   g_lim;

#undef      YY_INPUT
#define   YY_INPUT(b, r, ms) (r = my_yyinput(b, ms))
static   int   my_yyinput(char* buf,   int max);

// cmd-arguments related
#define   MAX_ARG_CNT   256
static   char*   g_argv[MAX_ARG_CNT];
static   int   g_argc = 0;

static   void   add_arg(const   char*   xarg);
static   void   reset_args();

// cmd-handlers
static   void   exec_simple_cmd();
%}
```

在 8.3.3 节的主函数中，命令行的输入信息被放在了全局变量 g_ptr 和 g_lim 中，这里进一步告诉 flex 如何使用该输入信息，通过宏 YY_INPUT 和函数 my_yyinput 来具体实现。

parse.lex 文件的第二部分描述了一组模式匹配规则，这是语法分析的核心。

```
%%
[^ \t\n]+       {add_arg(yytext);}
\n              {exec_simple_cmd();    reset_args();}
.                  ;
%%
```

第 2 行的意思是，如果扫描到一个不含空白字符和换行符的字符串，则将其加入到参数数组中去（函数 add_arg(yytext)的功能是将该字符串 yytext 放到参数数组 g_argv[]中去，详细代码见下文的第三部分）。第 3 行的意思是，如果扫描到换行符，则将参数数组中的所有参数构成的序列当作一个简单命令去执行，然后清空参数数组。第 4 行的意思是，忽略其他所有字符（在这里，因为不匹配前两项规则的字符只剩下空白字符，所以实际忽略的是所有空白字符）。

parse.lex 文件的第三部分是前面用到的所有函数的定义。

```
static    void    add_arg(const    char*    arg)
{
          char    *t;
          if ((t = malloc(strlen(arg) + 1)) == NULL) {
                    perror("Failed to allocate memory");
                    return;
          }
          strcpy(t, arg);
          g_argv[g_argc] = t;
          g_argc++;
          g_argv[g_argc] = 0;
}

static    void    reset_args()
{
          int    i;
          for (i = 0; i < g_argc; i++) {
                    free(g_argv[i]);
                    g_argv[i] = 0;
          }
          g_argc = 0;
}

1     static void exec_simple_cmd()
2     {
3               pid_t    childpid;
4               int    status;
5               if ((childpid = fork()) == -1) {
6                         perror("Failed to fork child");
```

```
7                   return;
8           }
9           if (childpid == 0) {
10                  execvp(g_argv[0], g_argv);
11                  perror("Failed to execute command");
12                  exit(1);
13          }
14          waitpid(childpid, &status, 0);
15  }

static int my_yyinput(char* buf, int max)
{
        int    n;
        n = g_lim - g_ptr;
        if (n > max)
        n = max;

        if (n > 0) {
                memcpy(buf, g_ptr, n);
                g_ptr += n;
        }
        return n;
}
```

其中，大部分函数的实现比较直接，这里不多解释。下面介绍如何执行一条简单命令，即函数 exec_simple_cmd()的实现过程，该函数中的行号是附加的。

8.3.5 简单命令的执行

函数 exec_simple_cmd()执行时，所有的参数（包括命令名）都已按序放在了数组 g_argv[]中，此函数的功能就是创建一个子进程，来运行这个命令，并传递相应的参数，然后等待该子进程结束。

在函数 exec_simple_cmd()的第 5 行，调用函数 fork()创建了一个子进程，随后子进程在作了第 9 行的条件判断后会执行第 10 行的函数调用 execvp()，此调用将会执行参数数组中的命令并传递参数。如果上述调用成功，那么第 11 行将没有机会执行；否则子进程将在第 11 行报错后于第 12 行结束。父进程则在第 9 行作了条件判断后转而执行第 14 行，以等待子进程结束，然后返回。

8.3.6 myshell的Makefile

最后来看本项目的 Makefile，如下所示：

```
CC=gcc
CFLAGS=-g    -Wall
LEXSRC=parse.lex
SRC=myshell.c    lex.yy.c
```

```
all:
        flex    $(LEXSRC)
        $(CC)   $(CFLAGS)   -o   myshell   $(SRC)   -lfl
```

命令 flex $(LEXSRC)生成程序文件 lex.yy.c，其中包含了语法分析函数 yylex()；最后一行命令编译主程序 myshell.c 和文件 lex.yy.c，并连接生成可执行文件 myshell。

8.3.7 实验指南

所有要求实现的功能都是 bash 的子集，所以若有不了解的功能试用 bash 就可以熟悉了。

改变当前目录可使用 chdir 函数；重定向的实现要使用 dup 函数；管道的实现可以使用 pipe 函数；这几个函数以及 myshell 中用到的函数 fork 和 execvp 的说明信息可以参考附录 B。

第 9 章　内核编程基础实验

9.1　实验 4——内核模块

模块能动态扩充 Linux 的功能而无须重新编译内核，Linux 内核的许多功能都是基于模块实现的。我们将在本实验中学习模块的基本概念和原理，学习内核模块编程的基本技术，然后利用内核模块编程访问进程描述符，操作内核的基本数据结构，从而加深对进程的理解。

9.1.1　实验内容

本实验由两个子任务组成：

（1）设计一个模块，该模块的功能是列出系统中所有内核线程的程序名、PID 号和进程状态。

（2）设计一个带参数的内核模块，其参数为某个进程的 PID 号，该模块的功能是列出该进程的家族信息，包括父进程、兄弟进程和子进程的程序名、PID 号。

9.1.2　Linux内核模块简介

Linux 内核是单体的（monolithic），即编译器把各个内核组件链接生成一个大的可执行文件。另一种内核结构是微内核（microkernel），它只把一些最基本的功能，如进程间通信、同步原语，做入内核，其他（如文件系统、存储器管理、设备驱动等）都作为用户态进程出现，相对普通的应用程序来讲，它们可以看成服务器进程，为应用程序提供服务。微内核有许多理论上的优势，如模块化更好、易于移植、更加稳定、不易崩溃等，但是在性能方面一直比不上单体内核，因为微内核体系导致的进程间通信开销非常大。

Linux 的内核模块（module）机制不仅可以弥补单体内核相对微内核的一些不足，而且对性能没有影响。内核模块是一个目标文件，可以动态载入内核，也可以动态卸载。实际上，Linux 中大多数设备驱动程序或文件系统都以模块方式实现，因为它们数目繁多，体积庞大，不适合直接编译在内核中。而通过模块机制，在需要使用它们的时候再临时加载，是最适合不过的。另外一个明显的好处是，当采用模块技术进行开发时，用户修改代码后只需重新编译加载模块，而不必重新编译内核和引导系统。

当一个模块被加载到内核中的时候，它就成为内核代码的一部分，与其他内核代码的地位完全相同。模块自身不是一个独立的进程，当前进程运行时调用到模块代码时，我们认为该段代码就代表当前进程在核心态运行。值得注意的是，由于模块中的代码与内核中其他部分的代码地位相同，因此模块中的一个代码错误就可能导致整个系统的崩溃。

9.1.3　内核符号表

模块编程可以使用内核的一些全局变量和函数，在将模块源程序编译成目标文件后这些内核符号依然是未决的（unresolved），直到模块加载时才进行动态解析，把对内核符号的引用转换成相应的地址。

内核维护了几个符号表（symbol table），这些表中包含模块可以访问的符号和它们相应的地址，用户可以通过/proc/kallsyms 文件查看。内核的全局变量或函数能进入这些符号表由模块访问。需要使用宏 EXPORT_SYMBOL 或 EXPORT_SYMBOL_GPL 来输出符号，EXPORT_SYMBOL_GPL 输出的符号只能被采用 GPL 许可的模块访问。以内核函数 printk 为例，在 kernel/printk.c 中有如下一行代码意味着所有的模块均可以使用该函数：

EXPORT_SYMBOL(printk);

模块也可以输出符号，当模块加载时，其输出的符号也加入内核符号表。新加载的模块可以使用已加载模块的输出符号，利用现成的代码完成更复杂的功能，这种现象称为模块栈（module stacking），这些模块像搭积木一样加载，但卸载时必须按“先进后出”的方式。

9.1.4 内核模块编程介绍

9.1.4.1 内核模块实例

我们选择一个简单的例子来说明内核模块的编写、编译和加载过程。下面是内核模块 hello.c 的代码：

```
1   #include <linux/init.h>
2   #include <linux/module.h>
3   MODULE_LICENSE("GPL");
4
5   static int hello_init(void)
6   {
7       printk(KERN_ALERT "Hello, world\n");
8       return 0;
9   }
10  static void hello_exit(void)
11  {
12      printk(KERN_ALERT "Goodbye\n");
13  }
14  module_init(hello_init);
15  module_exit(hello_exit);
```

为了编译该模块程序，在 hello.c 所在目录下面还要编写一个名为 Makefile 的文件，其内容如下：

```
17  ifneq ($(KERNELRELEASE),)
18      obj-m := hello.o
19  else
20      KDIR := /lib/modules/$(shell uname -r)/build
21      PWD := $(shell pwd)
22
23  default:
24      $(MAKE)  -C $(KDIR) M=$(PWD) modules
25  clean:
```

```
26          $(MAKE)  -C $(KDIR) M=$(PWD) clean
27      endif
```

键入如下命令：

#make

如果 make 命令执行成功，可以看到当前目录下生成了一些文件，如 hello.mod.c，hello.ko 等，这其中最重要的是 hello.ko 文件。接下来以超级用户身份使用下面一组命令来测试模块，过程如下：

 # insmod ./hello.ko

Hello, world

#lsmod | grep hello

hello　　1664　0

 # rmmod hello

Goodbye

其中，insmod 是加载模块命令，lsmod 用来列出系统已加载的模块，rmmod 用来卸载模块。

9.1.4.2　模块编程的基本知识

先简要介绍内核模块编程和用户态编程的区别。首先，内核模块编程不能使用 C 函数库，内核模块只能使用一些内核函数，以 hello 模块为例，输出信息时使用内核函数 printk，而不是标准库函数 printf。其次，内核模块代码运行在核心态，这意味着函数使用的栈是核心栈，该空间极为有限，一般是 4KB 或 8KB，所以不要定义占用空间很大的自动变量，如果确实需要使用大的结构，建议使用动态分配的空间。最后，内核代码不能使用浮点运算，这样做是为了节省开销。

hello 模块最前面两行包含两个内核头文件，Linux 内核的大部分头文件位于内核源代码 include 目录下，所以 include 目录是默认指定的。以 linux/init.h 来说，实际就是内核目录下的 include/linux/init.h 文件。宏 module_init 和 module_exit 的定义在 linux/init.h 中，但为什么要包含 linux/module.h 就不是十分明显了，实际上 hello 模块还调用了另外一个内核函数 printk，该函数的原型声明在 linux/kernel.h 中，但是 linux/module.h 间接包含了 linux/kernel.h，因为大部分实际内核模块都用到了 linux/module.h，出于介绍的目的，在此我们用 linux/module.h 取代了 linux/kernel.h。

内核模块程序里面没有 main 函数，每个模块都应该定义两个函数，一个函数用来初始化，常用来注册和申请资源，该函数返回 0，表示初始化成功（见示例第 8 行），其他值表示初始化失败；另一个函数用来退出，常用来注销和释放资源。一般这两个函数分别用宏 module_init 和 module_exit 来标识，module_init 标记的函数在加载模块时调用，module_exit 标记的函数在卸载模块时调用。对于 hello 模块来说，该模块仅有的两个函数 hello_init 和 hello_exit 便起上述作用，只是它们的功能极为简单，各自输出一行信息而已。

示例第 3 行调用 MODULE_LICENSE 告诉内核该模块采用 GPL 许可，其他允许的许可还包括"GPL v2"、"GPL and additional rights"、"Dual BSD/GPL" 、"Dual MIT/GPL"、"Dual MPL/GPL"，内核均认为以上许可是与 GPL 兼容的。 有些商业公司发布的模块仅以二进制方式分发，不提供源代码，这种情况下模块许可被认为"Proprietary"，内核开发者一般不愿意

对这种模块的用户提供技术帮助。如果未给模块指定许可，加载模块时很可能出现“module license 'unspecified' taints kernel”之类的信息。

示例第 7 和 12 行调用的内核输出函数 printk 原型如下：

asmlinkage int printk(const char *fmt, ...);

可以看到，printk 与标准 C 库函数 printf 几乎一样，实际上两者用法也类似。但是 printk 的 fmt 首部一般是如 KERN_ALERT 之类的优先级，如果没有优先级，则采用系统默认值。printk 允许的优先级范围从 0～7，值越低表示优先级越高，它们的符号标记依次是 KERN_EMERG， KERN_ALERT， KERN_CRIT， KERN_ERR， KERN_WARNING，KERN_NOTICE，KERN_INFO 和 KERN_DEBUG。优先级和系统设置决定了 printk 中的信息以何种方式输出，这其中牵涉较多内容，在此就不详细介绍了。如果用户在文本控制台环境下，printk 产生的信息可以直接显示，但如果用户在 X 图形界面下的仿真终端环境下，信息不会直接显示。无论如何，printk 信息一般会被添加到文件/var/log/messages 的尾部，所以总是可以通过如下命令获取 printk 的输出信息：

#tail /var/log/messages

在加载模块时还可以传递参数，模块参数通过宏 module_param 声明，该宏定义在文件 linux/moduleparam.h 中。module_param 带三个参数，第一个参数是变量名；第二个参数是变量类型，目前支持的有 int，charp（字符指针），long 等；第三个参数是许可标志位，目前设为 S_IRUGO 即可。稍微修改一下 hello.c 来说明如何控制模块参数。在第 2 行后添加新的一行：

#include <linux/moduleparam.h>

然后在示例原第 3 行后添加下面 4 行：

static char *my_string = "parameter";

static int my_int = 1;

module_param(my_string, charp, S_IRUGO);

module_param(my_int, int, S_IRUGO);

在示例原第 7 行后添加下面一行：

printk(KERN_ALERT "%s %d\n", my_string, my_int);

重新编译 hello.c 后使用如下命令行加载模块，然后观察模块产生的输出。

#insmod hello.ko my_string="string"my_int=27

9.1.4.3 Makefile介绍

下面介绍 Makefile 文件。第 17 行的 KERNELRELEASE 是在内核源码顶层 Makefile 中定义的一个变量，在第一次读取 Makefile 时，KERNELRELEASE 还没有定义，为空，所以 ifneq 判断为假，执行 else 之后的内容。第 20 行 KDIR 是定位内核代码目录，如果查看 /lib/modules/2.6.xxx/build 文件就会发现，该文件是一个符号链接，指向内核代码树。第 21 行 PWD 指向当前目录。假如 make 的目标是 clean，则清除当前目录下各个生成文件，然后结束。当 make 的目标为 default 时，-C $(KDIR)指明跳转到内核源码目录下读取那里的 Makefile，M=$(PWD)表明返回到当前目录再次处理该目录下的 Makefile。此时 KERNELRELEASE 已被定义，第 17 行的结果为真，第 18 行得到执行，该行实际是 2.6 内核构建系统 kbuild 的语法，指示生成目标模块 hello.ko 时依赖于目标文件 hello.o，当然 hello.c 最终依赖于 hello.o，

其中的细节无须我们关心。

如果模块 hello.ko 的生成来自两个 C 文件 hello1.c 和 hello2.c，则 Makefile 相应的行应改为

```
obj-m := hello.o
hello-objs := hello1.o hello2.o
```

9.1.5 实验指南

Linux 进程描述符 task_struct 结构定义在 linux/sched.h 文件中，该结构包含众多的成员项，下面仅列出与本实验相关的成员项：

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_STOPPED    4
#define TASK_TRACED     8
#define EXIT_ZOMBIE             16
#define EXIT_DEAD       32
#define TASK_NONINTERACTIVE     64
#define TASK_DEAD       128
struct task_struct {
    volatile long state;                    /*进程的运行状态 */
    struct list_head tasks;                 /*线程组长链表，是结点*/
    struct mm_struct *mm;                   /*描述进程地址空间*/
    pid_t pid;                              /*进程 ID 号*/
    pid_t tgid;                             /*线程组 ID 号*/
    struct task_struct *real_parent;        /* 真正的父进程 */
    struct task_struct *parent;             /*父进程,非调试情况等同 real_parent */
    struct list_head children;              /*子进程链表，是表头*/
    struct list_head sibling;               /*兄弟进程链表，是结点*/
    char comm[TASK_COMM_LEN];               /*程序名，不包含路径*/
}
```

Linux 的进程和轻量级进程/线程均有相应的 task_struct 结构和唯一的 PID 号，而 POSIX 要求同一组的线程有统一的 PID，为此 Linux 引入了 tgid（thread group identifier），同一个组的各线程的 pid 值不同，但是它们的 tgid 值相同，tgid 值实际是线程组第一个线程的 pid 值，该线程称为线程组长（thread group leader）。对于普通进程来讲，其 pid 和 tgid 是一样的，也可以看成是线程组长。此外，Linux 系统的进程包含一种特殊的类型——内核线程（kernel thread），它完成内核的一些特定任务，并始终在核心态运行，没有用户态地址空间，所以相应的 task_struct 结构的 mm 成员为 NULL。

Linux 采用多个链表确保有效查找系统里的进程，双向链表结构 list_head 在 Linux 内核中广泛使用，而不仅仅用于组织进程，下面是它的定义：

```
struct list_head {
    struct list_head *next, *prev;
```

};

因为 list_head 一般嵌入到内核数据结构中，为了通过 list_head 的 prev，next 成员方便地访问到相应的内核数据结构，内核提供了一系列的宏来实现链表的常规操作，感兴趣的读者可以参看 linux/list.h 文件的相关实现。

本次实验只涉及读取已经建立好的链表，其他的操作，如删除、插入等均不涉及，我们通过下面的例子说明如何遍历链表元素。假定有如下结构：

```
struct kern_data1{
   .....
   list_head list1;
   ......
};
struct kern_data2{
   .....
   list_head list2;
   ......
};
struct kern_data1 data1;
struct kernel_data1 *pdata = &data1;
```

其中，kern_data1 和 kern_data2 两个结构类型可能不同，也可能相同，即使相同，list1 和 list2 也不一定是同一个成员。kern_data1 的 list1 是一个链表的表头，该链表的每个元素的类型均是 struct kern_data2。kern_data2 类型的对象通过成员 list2 加入 data1.list1 链表，如图 9-1 所示。如果 data1.list1 链表已经建立，可通过如下方式访问该链表的各项：

```
struct list_head *p;
struct kernel_data2 *my;
list_for_each(p, &pdata->list1) {
      my = list_entry(p, struct kernel_data2, list2);
      //下面可以对 my 指向的 kern_data2 对象操作
}
```

宏 list_for_each(p, &pdata->list1)遍历 pdata->list1 链表，每次 p 指向一个 kern_data2 对象的 list2 成员，而宏 list_entry(p, struct kernel_data2, list2)进一步得到 kern_data2 对象的指针。

在了解了内核通用链表后，下面提供一些对完成实验内容有用的信息。

子任务（1）可以利用内核的线程组长链表，每个线程组长通过 task_struct 结构的 tasks 成员加入该链表。值得庆幸的是，内核专门提供的宏 for_each_process 可以访问该链表中的每个进程，用法如下：

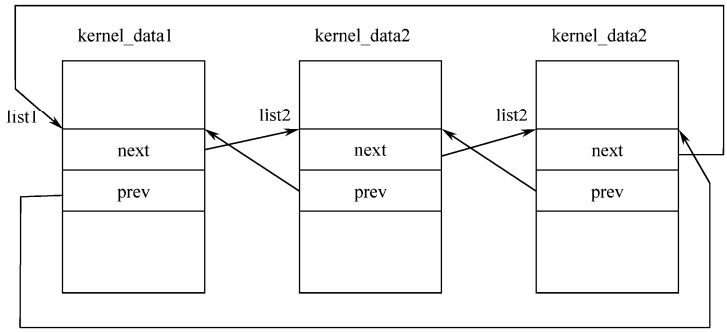

图 9-1　内核通用双向链表

```
struct task_struct * p;
for_each_process(p) {
    //对 p 指向的进程描述符操作
}
```

子任务（2）需要了解 Linux 进程家族的组织情况。来自参考文献[3]的图 9-2 很好地描述了这种关系，子进程通过 task_struct 的 sibling 成员加入父进程的 children 链表。对子进程链表和兄弟进程链表的访问，都可以通过宏 list_for_each 和 list_entry 来实现。对于指定的 pid，可以通过宏 find_task_by_pid 找到其相应的 task_struct 结构，宏 find_task_by_pid 和 for_each_process 均在文件 linux/sched.h 中定义。

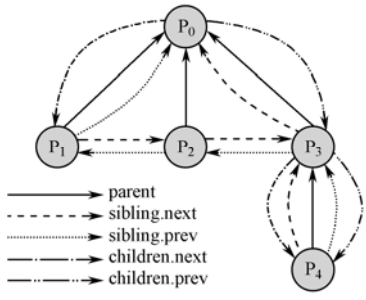

图 9-2　进程家族关系

9.1.6　测试

实际上通过用户态的 ps 命令可以观察到本次实验的结果，因此模块输出和 ps 命令的结果可互相印证。例如观察内核线程，可使用如下命令：

$ps aux | less

如果发现进程的程序名用中括号包围，则该进程实际是内核线程，如[ksoftirqd/0]。系统中的进程树则可以通过如下命令观察：

#ps axjf | less

9.2 实验 5——proc文件系统编程

我们在 8.2 节实验 2 中已经接触到了 proc 文件系统的基本概念，并且编写程序读取了其中几个文件的内容，扫除了对 proc 文件系统的神秘感。在本次实验中，我们将改进实验 4 的不足之处，利用内核模块编程在/proc 目录中增加若干文件，用户通过对这些文件的读/写来访问模块提供的功能。通过本次实验，读者不仅能够学习到 proc 文件的创建方法，而且也将掌握了一种用户态和核心态通信的方法。

9.2.1 实验内容

在实验 4 中，虽然我们可以通过带参数的模块获取进程 p1 的家族信息，但是我们想获得另外一个进程 p2 的家族信息就不太方便了，我们只能先卸载该模块，然后以进程 p2 的 pid 为参数重新加载该模块。此外，如果使用 printk 产生输出信息，这些信息和系统其他信息混杂在一起，不利于程序自动提取分析。

本次实验我们仍然实现实验 4 子任务（2）的功能，但不使用模块参数的方式，而是通过 proc 文件系统实现用户态和核心态通信，进程的 pid 由 proc 文件传入，进程家族信息也通过 proc 文件访问。

9.2.2 proc文件系统编程简介

类似于实验 4，我们先给出一个可运行的实例，然后再讲述。

9.2.2.1 proc文件系统编程示例

下面是示例 procfs_example.c 的源代码：

```
1   #include <linux/init.h>
2   #include <linux/module.h>
3   #include <linux/proc_fs.h>
4   #include <linux/sched.h>
5   #include <asm/uaccess.h>
6   MODULE_LICENSE("GPL");
7
8   #define BUFSIZE   1024
9
10  static char global_buffer[BUFSIZE];
11  static struct proc_dir_entry *example_dir, *hello_file, *current_file, *symlink;
12
13  static int proc_read_current(char *page, char **start, off_t off, int count,
14                               int *eof, void *data)
15  {   //当用户读取"current"文件的内容时，内核调用这个函数
16      int len;
17      len = sprintf(page,"current process-->name:%s gid:%d pid:%d\n",
18                    current->comm, current->signal->pgrp, current->pid);
19      return len;
```

```
}

static int proc_read_hello(char *page, char **start,off_t off, int count,
                           int *eof, void *data)
{    //当用户读取"hello"文件的内容时，内核调用这个函数
     int len;
     len = sprintf(page, "%s", global_buffer);
     return len;
}

static int proc_write_hello(struct file *file,const char *buffer,
                            unsigned long count,void *data)
{    //当用户写"hello"文件的内容时，内核调用这个函数
     int len;
     if (count > BUFSIZE - 1)     //如果写内容过长，则截断
          len = BUFSIZE - 1;
     else
          len = count;
     //从用户的缓冲区获得要写的数据
     if (copy_from_user(global_buffer, buffer, len))
          return -EFAULT;
     global_buffer[len] = '\0';
     return len;
}

static int proc_init(void)
{
     example_dir = proc_mkdir("proc_test", NULL);//创建目录  /proc/proc_test
     example_dir->owner = THIS_MODULE;
     //创建只读文件    /proc/proc_test/current
     current_file = create_proc_read_entry("current", 0444, example_dir,
                                           proc_read_current, NULL);
     current_file->owner = THIS_MODULE;
     //创建一个可读/写的文件/proc/proc_test/hello
     hello_file = create_proc_entry("hello", 0644, example_dir);
     strcpy(global_buffer, "hello");    //预设文件内容为 hello
     hello_file->read_proc = proc_read_hello;
     hello_file->write_proc = proc_write_hello;
     hello_file->owner = THIS_MODULE;
     //创建一个链接  /proc/proc_test/current_too，指向 current
     symlink = proc_symlink("current_too", example_dir,"current");
     symlink->owner = THIS_MODULE;
     return 0;
}

static void proc_exit(void)
{
     remove_proc_entry("current_too", example_dir);
```

```
68          remove_proc_entry("hello", example_dir);
69          remove_proc_entry("current", example_dir);
70          remove_proc_entry("proc_test", NULL);
71      }
72      module_init(proc_init);
73      module_exit(proc_exit);
```

上面的程序实际上是一个内核模块，按照实验 4 介绍的方法编译生成 procfs_example.ko 文件，然后使用 insmod 命令加载，我们知道 proc_init 函数将被调用。proc_init 首先在/proc 目录下创建子目录 proc_test，然后在这个目录下创建两个文件 current 和 hello。程序还创建了一个符号链接文件 current_too，读 current_too 的效果跟读 current 是一样的。下面是笔者机器上的一组测试命令和结果：

#cat /proc/proc_test/current

current process-->name:cat gid:7249 pid:7249

 # cat /proc/proc_test/hello

hello

 # echo "1234" > /proc/proc_test/hello

 # cat /proc/proc_test/hello

1234

 # cat /proc/proc_test/current_too

current process-->name:cat gid:7272 pid:7272

current 是只读的，它显示当前进程的 pid，gid 和可执行程序名等信息，所以第一次执行 cat /proc/proc_test/current 就是运行 cat 命令，产生进程的信息，然后该进程就结束了。而后面执行 cat /proc/proc_test/current_too 输出的是再次运行 cat 命令所产生的进程信息，所以两次信息并不一样。hello 文件是可读/写的，其内容被初始化为“hello”，见代码第 55 行。

这里我们再次强调模块代码和进程的关系，模块不是进程，模块加载后就变成内核的一部分，进程访问内核服务时可能会执行模块的某些代码，如前面的 4 个命令行（三个 cat 和一个 echo）都运行了模块的部分代码。

最后，可以卸载该模块，proc_exit 函数被调用，我们将发现前面创建的 proc 目录和文件都消失了。

9.2.2.2 proc文件系统的核心数据结构

与磁盘文件系统不同的是，proc 文件系统的信息驻留在内存中，这意味着 proc 文件系统不能占太多空间，有两个因素确保了这一点，一是 proc 文件系统自身规模很小，二是 proc 文件（甚至包括一些 proc 目录）的内容是根据用户临时需要动态生成的，不需要永久性地占用内存。

proc 文件系统保存在内存中的元数据用 struct proc_dir_entry 结构描述，它既可以描述目录也可以描述文件，定义如下：

```
struct proc_dir_entry {
    unsigned int low_ino;
```

```
    unsigned short namelen;                 /*名字的长度*/
    const char *name;                       /*结点的名字，也就是该 proc 文件的名字*/
    mode_t mode;                            /*文件的类型和权限*/
    nlink_t nlink;                          /*文件的链接数*/
    uid_t uid;
    gid_t gid;
    unsigned long size;
    struct inode_operations * proc_iops;
    struct file_operations * proc_fops;
    get_info_t *get_info;
    struct module *owner;                   /*该文件的拥有者模块*/
    struct proc_dir_entry *next, *parent, *subdir; /*见下面的描述*/
    void *data;
    read_proc_t *read_proc;                 /*读操作函数*/
    write_proc_t *write_proc;               /*写操作函数*/
    atomic_t count;                         /*引用计数*/
    int deleted;                            /*删除标志*/
};
```

proc 文件系统实际可看成各个结点均是 proc_dir_entry 的树形结构，每个结点都保存了其父结点（parent 成员）、孩子结点（subdir 链表）和兄弟结点（next 成员）的信息，便于维护管理。

对于 proc 编程人员来说，创建和初始化 proc_dir_entry 的工作基本可借助内核 API 完成，只有少部分 proc_dir_entry 的成员项需要程序员手工控制，后面我们将看到这一点。

9.2.2.3 proc文件系统编程接口

下面介绍几个内核函数，通过这些函数可以请求内核在 proc 文件系统中创建或删除文件/目录，这些函数的原型在 linux/proc_fs.h 文件中。

（1）创建目录——proc_mkidr()

函数原型为

```
struct proc_dir_entry    *proc_mkdir(const char *name, struct proc_dir_entry *parent)
```

该函数将创建一个目录，目录名为 name，父目录为 parent。如果要在根目录下创建子目录，则指定 parent 为 NULL。如果函数创建失败，返回 NULL，否则返回新建 proc_dir_entry 项的地址。用法见示例第 47 行。特别要指出的是，为了节省篇幅，我们在示例代码中忽略了对这些函数返回值的检查和错误处理，而这些对保证系统的健壮性是极其重要的。

（2）创建文件——create_proc_entry()

函数原型为

```
struct proc_dir_entry    *create_proc_entry(const char *name, mode_t mode,struct proc_dir_
                                            entry *parent)
```

该函数将创建一个 proc 文件/目录，名字为 name，文件类型和访问权限为 mode，其父目录为 parent。如果想在 proc 文件系统的根目录下创建，则指定参数 parent 为 NULL。如果函数创建失败，返回 NULL，否则返回新建 proc_dir_entry 项的地址。用法见示例第 54 行。注意，创建的文件和目录不能用常规文件系统的 rm，rmdir 命令删除，只能用下文介绍的

remove_proc_entry 来删除。

（3）创建只读文件——create_proc_read_entry()

函数原型为

struct proc_dir_entry *create_proc_read_entry(const char *name, mode_t mode, structproc_dir_entry *base, read_proc_t *read_proc, void * data)

该函数创建一个只读的 proc 文件，其实它只是简单地调用 create_proc_entry，并将返回结构的 read_proc 域的值置为 read_proc，data 域置为 data。如果函数创建失败，返回 NULL，否则返回新建 proc_dir_entry 项的地址。用法见示例第 50 行。

（4）创建符号链接——proc_symlink()

函数原型为

struct proc_dir_entry *proc_symlink(const char *name, struct proc_dir_entry *parent, char *dest)

该函数在 parent 目录下创建一个名字为 name 的符号链接文件，链接的目标是 dest。用法见示例第 60 行。

（5）删除文件/目录——remove_proc_entry()

函数原型为

void remove_proc_entry(const char *name, struct proc_dir_entry *parent)

该函数删除在 parent 目录下名为 name 的 proc 结点。如果要删除的文件正在使用中，置 proc_dir_entry 结构中的 deleted 标志，否则直接删除。用法见示例第 67～70 行。

（6）读/写文件接口——read_proc 和 write_proc

仅创建一个 proc 文件是不够的，文件还要有内容，要么通过该文件将内核信息提供给用户，要么用户通过写该文件影响内核行为。这里我们介绍最简单的处理方法，更复杂的技术手段可见参考文献[8]。

最简单的处理方法是用户自己提供 proc_dir_entry 结构的 read_proc 和 write_proc 成员。这两个函数都是回调函数，也就是说，当对该文件进行读/写时，系统会自动调用它们。用法见示例第 51 行的 proc_read_current，第 56 行的 proc_read_hello，第 57 行的 proc_write_hello。下面讨论如何实现用户自己的 read_proc 和 write_proc。

读文件接口函数的原型如下：

int (*read_proc)(char *page, char **start, off_t off, int count, int *eof, void *data);

read_proc 实际上有三种不同的实现方式，感兴趣的读者可以查看内核函数 proc_file_read 的源代码实现，从中得到详细的信息。三种方式中最简单的一种实现是除了参数 page 之外，忽略 read_proc 的所有参数，将文件的所有内容都写入参数 page 指向的缓冲区，然后返回文件的长度即可。这种实现方式必须有一个前提，就是 proc 文件的数据量很小，不会超过一个页面。

以/proc/proc_test/current 相应的读函数 proc_read_current 的实现为例，因为它只读取当前进程的 pid，gid 和可执行程序名，信息量绝对不会超过一个页面，所以可以采用上面提到的实现方式。顺便解释一下示例的第 17 和 18 行，sprintf 是核心态函数，声明在 linux/kernel.h 中，但它的用法和标准库函数一样；current 是一个宏，指向当前进程的 task_struct，使用它需要加头文件 linux/sched.h。

写文件接口函数的原型如下：

int (*write_proc)(struct file *file, const char *buffer,unsigned long count, void *data);

该函数将 buffer 开始的缓冲区中的 count 个字节写入 file 中。data 是私有数据，一般不需要关心。由于 buffer 一般是用户空间的指针，指向用户空间的缓冲区。因此应先调用 copy_from_user 将数据复制到内核空间中来。这里介绍两个内核函数 copy_to_user 和 copy_from_user，其原型如下：

```
unsigned long copy_to_user(void _ _user *to,   const void *from, unsigned long count);
unsigned long copy_from_user(void *to,   const void _ _user *from, unsigned long count);
```

使用时要包含头文件 asm/uaccess.h，前者将内核空间地址 from 开始的 count 个字节复制到用户空间指针 to 所指向的缓冲区，后者将用户空间地址 from 开始的 count 个字节复制到内核空间指针 to 所指向的缓冲区。

示例/proc/proc_test/hello 文件的写函数 proc_write_hello 非常简单，hello 文件使用缓冲区 global_buffer 保存数据，所以写文件 hello 不过就是调用 copy_from_user 将用户态数据复制到 global_buffer 而已。另外值得注意的是，write_proc 的参数里面根本没有文件偏移，所以用户态程序调用 write 函数向/proc/proc_test/hello 写数据时要一次性写完，如果多次调用 write，后面写的数据将覆盖前面的数据。

最后，解释示例的 52 行和 58 行。proc_dir_entry 结构的 owner 成员指向拥有该数据结构的模块，该成员用来阻止模块在使用中就被卸载，一般初始化为宏 THIS_MODULE，该宏定义在 linux/module.h 中。

9.2.3 实验指南

如果认真掌握了上面的示例，完成本次实验是一件比较容易的事情。仿照示例用模块来完成本任务，先在/proc 目录下面建立一个目录 process_family，然后在 process_family 建立两个文件 pid 和 family，文件 pid 可读可写，该文件用来控制下面的静态全局变量：

```
static pid_t pid = 1;   /*  默认进程为 init*/
```

文件 family 只读，该文件用来显示进程号为 pid 的家族信息。完成模块后编译加载，使用下列命令来测试模块功能：

```
#cat   /proc/process_family/pid
1
#cat /proc/process_family/family
[输出略]
#echo 3 > /proc/process_family/pid
#cat /proc/process_family/pid
3
#cat /proc/process_family/family
[输出略]
```

9.3 实验 6——编译内核及增加Linux系统调用

我们不仅可以通过模块来扩充内核的功能，而且也可以直接修改 Linux 内核代码来增加内核功能。修改后的内核代码必须重新编译生成新的映像文件，然后加载该映像文件便可以启用新内核。编译 Linux 内核是内核开发的必要基本功。

9.3.1 实验内容

本次实验由两部分组成。

第一部分仅仅要求编译一个干净的内核且加载成功，并不需要对内核修改。

第二部分是修改已经编译成功的内核，为其增加新的系统调用，扩充系统服务，提供给用户使用。

在此我们提供两个不同的系统调用功能说明，第一个较为简单，第二个相对复杂一些，读者可选择其一实现。下面是系统调用功能的具体要求。

（1）实现系统调用 psta，获取进程的若干信息。其原型如下：

int psta(struct pinfo *buf);

参数 buf 指向一个缓冲区，用于存放进程信息。结构 pinfo 定义如下：

```
struct pinfo {
        int nice;               /*  进程的 nice 值  */
        pid_t pid;              /*  进程 ID */
        uid_t uid;              /*  进程拥有者的用户 ID */
};
```

系统调用成功时的返回值为 0，系统调用失败时应返回错误码 EFAULT，表示 buf 指向非法地址空间。错误码定义见内核头文件 asm-generic/errno 和 asm-generic/errno-base.h。

（2）实现系统调用 noexec，用它设置进程允许执行 exec 系统调用的次数。该系统调用能用来防止一些缓冲区溢出攻击，这些攻击最终通过 exec 产生一个 shell 界面。

其原型如下：

int noexec(int nexec);

参数 nexec 含义如下：

- nexec = = 0，执行 exec 将失败。
- nexec = = n，n>0，可以执行 n 次 exec。
- nexec = = –1，执行 exec 的次数没有限制 。
- nexec = = 其他值， 不改变允许执行 exec 的次数 。

系统调用的返回值是本次 noexec 调用之前允许 exec 系统调用的次数。

新创建的子进程允许执行 exec 的次数继承自父进程，只有超级用户能够通过 noexec 增加可执行 exec 的次数。

9.3.2 Fedora下编译内核

这里以 Fedora Core 7 为例，讲述编译内核的过程，Fedora 其他版本的内核编译过程与之类似。如果读者采用不同的 Linux 发行版本，如 Ubuntu，Debian 等，编译内核过程可能会有所不同，请自行查阅相关资料。

9.3.2.1 第 1 步——下载内核

首先要决定编译哪个版本的内核，一般情况下，待编译的内核（以下一律称新内核）的版本不低于当前正在运行的内核版本，如果两者版本差距较大，则可能需要更新如 gcc，binutils 和 modutils 之类的编译工具。对本书所有需要编译内核的实验来说，只需要采用 Linux 发行版所对应的内核版本即可，这样所有内核编译工具都是现成的，无须更新，无疑减少了不必要的麻烦。先查看当前环境下的内核版本号：

#uname –r

2.6.21-1.3194.fc7

可见内核版本号是 2.6.21，到网址 http://www.kernel.org/pub/linux/kernel/v2.6/ 下载相应的内核压缩包（笔者选择的是 linux-2.6.21.7.tar.gz）到某个目录，例如/usr/src，然后输入如下命令解压生成代码树/usr/src/linux-2.6.21.7：

#cd /usr/src

#tar zxvf linux-2.6.21.7.tar.gz

#cd linux-2.6.21.7

9.3.2.2 第 2 步——生成内核配置文件.config

Linux 内核代码非常庞大，适用于许多体系结构，包含了大量驱动程序。用户在生成内核的时候要根据实际情况进行配置。所有的配置将保存在内核代码树顶级目录下的一个名为.config 的配置文件中。生成正确的配置文件是内核编译过程中至关重要的一步。

配置文件可以从头开始生成，但是这没有必要。因为当前正在运行的内核已经有对应的配置文件，该文件在/boot 目录下，利用它作为新内核配置文件的模板无疑是更好的做法。因此我们把该配置文件复制到/usr/src/linux-2.6.21.7 目录下，命令如下：

#make mrproper

#cp /boot/config-`uname -r`/.config .

第一个命令“make mrproper”用来保证内核树是干净的，如果内核树已经编译过，该命令有效，如果内核树是第一次编译则可以省略该命令。

现在虽然有了模板，但是.config 文件的配置并不一定囊括了新内核的所有编译选项（因为新内核可能是更新的版本，如 2.6.23），可以使用下面的命令：

#make oldconfig

该命令读取.config 文件并根据新内核版本更新它。具体过程是这样的，该命令输出新内核所有的配置项，如果配置项已经在.config 中有设置，则输出设置值；如果是新项，程序会停下来要求用户输入设置值，值的具体含义见附录 C。用户输入值后程序继续运行直到所有配置项处理完毕。用户也可以使用如下命令：

#make silentoldconfig

该命令的功能和“make oldconfig”相似，不过它不输出信息，除非是新选项需要用户输入的时候。

到现在为止，生成的.config 文件就可以使用了，进入第 3 步。但是也可以在此基础上进一步调整，参见附录 C。

9.3.2.3 第 3 步——编译和安装新的内核

在编译内核之前，还可以定义用户自己的内核版本号，这样做是为了便于识别。在内核代码树的根目录下有文件 Makefile，它的前 4 行是：

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 21
EXTRAVERSION = .7
```

把第 4 行改成 EXTRAVERSION = .7-foo，这样新内核版本号就是 2.6.21.7-foo。

然后执行下面三个命令:

```
#make all
#make modules_install
#make install
```

“make all”将生成期望的内核映像及模块。“make modules_install”将安装模块到“默认目录/lib/modules/<内核版本号>”下面。“make install”最终将内核映像等几个文件复制到“/boot”目录，并修改引导程序的配置以启用该新内核。

如果上述三个命令均执行成功，可以观察到引导程序 grub 的配置文件/boot/grub/menu.lst 的文件内容：

```
default=1
timeout=10
splashimage=(hd0,10)/boot/grub/splash.xpm.gz
hiddenmenu
title Fedora (2.6.21.7-foo)
    root (hd0,10)
    kernel /boot/vmlinuz-2.6.21.7-foo ro root=LABEL=/ rhgb quiet
    initrd /boot/initrd-2.6.21.7-foo.img
title Fedora-base (2.6.21-1.3194.fc7)
    root (hd0,10)
    kernel /boot/vmlinuz-2.6.21-1.3194.fc7 ro root=LABEL=/ rhgb quiet
    initrd /boot/initrd-2.6.21-1.3194.fc7.img
```

为了以后能直接操作菜单，把 hiddenmenu 那一行注释掉（行的最前面加一个“#”字符即可）或删除。然后使用 reboot 命令重启系统：

```
#reboot
```

如 图 9-3 所示，重启后可以看到GRUB菜单已经包含了新编译的内核，选择该选项启动即可尝试新内核。如果新内核启动失败，一般是因为配置有问题或工具集有问题，选择原来的内核启动，然后修订配置文件或更新工具集，再次编译内核。

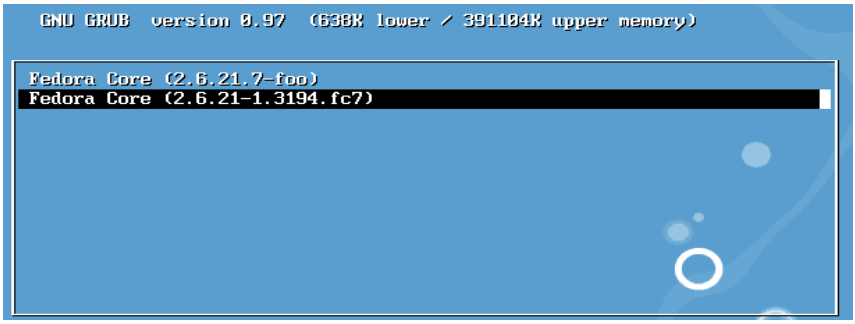

图 9-3　添加新内核后的 GRUB 菜单

9.3.3　添加psta系统调用

我们在 6.3 节提到过，若系统调用名为 xxx，则内核对应的实现函数名一般为 sys_xxx，据此我们把 psta 系统调用对应的内核函数命名为 sys_psta。下面假定当前工作目录是 /usr/src/linux-2.6.21.7，给出添加系统调用 psta 的基本过程。

（1）在文件 arch/i386/kernel/syscall_table.S 的尾部加上要新增的系统调用函数名称，如阴影行所示，注释中 320 表示它的系统调用号。

```
.long sys_tee                   /* 315 */
.long sys_vmsplice
.long sys_move_pages
.long sys_getcpu
.long sys_epoll_pwait
.long sys_psta                  /* 320 */
```

（2）在 include/linux 目录下添加头文件 psta.h，内容如下：

```
#ifndef _LINUX_PSTA_H
#define _LINUX_PSTA_H
struct pinfo{
        int nice;
        pid_t pid;
        uid_t uid;
};
#endif
```

在 kernel 目录下新建文件 psta.c，在该文件中实现 sys_psta 函数。

```
#include <linux/linkage.h>
#include <linux/psta.h>
……
asmlinkage    int sys_psta(struct pinfo *buf)
{
```

```
    ……
}
```

宏 asmlinkage 定义在文件 linux/linkage.h 中，表示函数的参数通过栈传递，而不是通过寄存器，所有系统调用都遵循这种参数传递方式。

sys_psta 的实现非常简单，无非就是把 task_struct 结构中的几个成员复制到用户态空间。值得一提的是，nice 表示进程的优先级，取值范围为[–20，19]，数值越低表示优先级越高。内核没有直接存储 nice 值，而是通过一个简单的变换后将它存放在 task_struct 结构的 static_prio 成员中。两者之间的转换关系见下面几个宏（位于文件 kernel/sched.c 中），其中 MAX_RT_PRIO 值为 100。

```
#define NICE_TO_PRIO(nice)      (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio)      ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p)            PRIO_TO_NICE((p)->static_prio)
```

（3）修改文件 kernel/Makefile，使 psta.c 能在内核编译时可见。Makefile 中有如下一行：

```
obj-y        = sched.o fork.o exec_domain.o panic.o printk.o profile.o \
```

添加 psta.o 到该行中：

```
obj-y        = psta.o sched.o fork.o exec_domain.o panic.o printk.o profile.o \
```

值得说明的是，第 2 步中 sys_psta 的实现不一定要放在一个新文件中，例如文件 kernel/sys.c 也许就是添加 sys_psta 系统调用的合适位置，这样第 3 步就没有必要了。

（4）在 include/asm-i386/unistd.h 里面加上系统调用号的宏定义，在该文件中有如下几行：

```
#define __NR_epoll_pwait    319

#ifdef __KERNEL__

#define NR_syscalls 320
```

可以看到，按照惯例系统调用号的宏名以“__NR_”开头，而其后跟着的数值则是系统调用号。此外，NR_syscalls 表示的值应该是最大的系统调用号加一。所以修改后的内容如下：

```
#define __NR_epoll_pwait    319
#define __NR_psta           320

#ifdef __KERNEL__

#define NR_syscalls 321
```

关于 include/asm-i386/unistd.h 要特别提示，我们假定当前平台是 IA32。如果我们的机器是基于安腾处理器的，则 asm-i386 要换成 asm-ia64。因为 Linux 支持不同的体系结构，为了保证代码可移植，Linux 假定 include 目录含有一个子目录 asm，它实际是一个符号链接，指向当前具体的架构，如 asm-i386，内核的其他代码只需要访问 asm 目录。遗憾的是，asm 符

号链接是在内核编译过程中根据具体平台产生的，目前并不存在，所以我们直接访问asm-i386。

（5）修改 include/linux/syscalls.h，加上函数 sys_psta 的声明。在该文件的首部添加一行：

#include <linux/psta.h>

在该文件的最后一行“#endif”之前添加一行：

asmlinkage int sys_psta(struct pinfo *buf);

（6）重新编译内核。这里特别要提醒初学者，因为我们已经有了正确的.config，最好备份一份到别的目录下以防被删除。上一次编译内核时已经在内核目录下生成了许多中间文件，所以本次编译内核之前要删除这些文件。这可以使用如下命令：

make mrproper

该命令连.config 文件都会删除，所以等命令执行完后需要把备份的.config 文件复制回来，然后执行前面的第 3 步就可以了。因为新内核包含了用户自己的代码，所以很可能会在“make all”时因编译出错而停止，根据错误提示信息修改错误后，可以重复本步骤。

9.3.4 测试新增系统调用psta

在使用新内核引导系统后，需要编写应用程序来测试新增系统调用的功能。遗憾的是，新增的系统调用没有对应的 glic 库函数。尽管如此，glic 库提供了函数 syscall，以间接方式让用户能方便地使用新系统调用，该函数原型如下：

#include <sys/syscall.h>

#include <unistd.h>

int syscall(int number, ...);

syscall 的第一个参数是系统调用号，后面的参数是该系统调用的各个参数。syscall 执行由系统调用号所指定的系统调用，返回值就是系统调用返回值。下面给出一个示例：

```
#include <sys/syscall.h>
#include <unistd.h>
#include <assert.h>

int main(void)
{
    /* __NR_write 为 write 系统调用编号
    * syscall(__NR_write, 1, "hello\n", 6)等价于 write(1, "hello\n", 6)
    */
    assert(6 = = syscall(__NR_write, 1, "hello\n", 6));
    return 0;
}
```

如果 glic 版本早于 Linux 版本发布，则 glibc 不可能有对应于新系统调用的库函数，这时候可以通过 syscall 以间接方式来执行新的系统调用。

下面给出 psta 系统调用的测试程序：

```
#include <unistd.h>
```

```
#include <sys/syscall.h>
#include <assert.h>
#include <errno.h>

struct pinfo{
        int nice;
        pid_t pid;
        uid_t uid;
};
int main(void)
{
    struct pinfo info;
    int ret;

    ret = syscall(320, &info);
    assert(ret = = 0);
    assert(info.nice = = nice(0));
    assert(info.pid = = getpid());
    assert(info.uid = = getuid());

    ret = syscall(320, NULL);
    assert(ret = = -1 && errno = = EFAULT);
    return 0;
};
```

在内核版本 2.6.20 以前，还有一种常见的方法用于执行新增的系统调用，文件 incude/asm/unistd.h 中有_syscall0～_syscall6 等 7 个宏提供类似 syscall 函数的功能。但是从版本 2.6.20 起这些宏已经被去掉了，用户想使用它时可以从老版本的文件中复制出来使用，具体的用法可查阅相关资料。

9.3.5 noexec系统调用的实现

相比 psta 系统调用，noexec 的实现更复杂。为了鼓励读者独立完成任务的积极性，我们不给出详细的实现细节。这里仅提出几个与实现有关的问题供思考：

（1）为了跟踪允许执行次数，是否要修改进程描述符 task_struct？

（2）为了保证继承语义，最早的进程应该如何初始化？

（3）fork 系统调用是否要做修改？

（4）execve 系统调用是否要做修改？

如果上面几个问题都能正确回答，实现 noexec 也就没有技术障碍了。此外，noexec 系统调用的测试程序也比 psta 系统调用的测试程序复杂一些。

第 10 章　内核编程综合实验

10.1　实验 7——进程隐藏

前面的几个内核编程实验难度都不大，跟内核已有的代码交互很少。本次实验涉及面相对较广，综合了前面所有的知识点，并加入了一个处理内核编译选项的环节。本实验要求在原有的基础上，进一步了解 proc 文件系统的实现机制和 VFS 的一些机制，以便巩固和提高读者的内核编程能力。

10.1.1　实验内容

实现一个系统调用 hide 来隐藏进程，使用户无法使用 ps 或 top 命令观察到进程状态。要求实现下面 7 项基本功能。

（1）要求实现系统调用 int hide(pid_t pid, int on)，在进程 pid 有效的前提下，如果 on 置 1，进程被隐藏，用户无法通过 proc 文件系统观察到进程状态；如果 on 置 0 且此前为隐藏状态，则恢复正常状态。系统调用的返回值请读者自行设计，但要合理。

（2）考虑权限问题，只有根用户才能隐藏进程。

在完成上述功能的前提下，可以选做下面的扩展功能，其中（4）和（5）择一，（6）和（7）择一。

（3）设计一个新的系统调用 int hide_user_process(uid_t uid, char *binname)，参数 uid 为用户 ID 号，当 binname 参数为 NULL 时，隐藏该用户的所有进程；否则，隐藏二进制映像名为 biname 的用户进程，系统调用的返回值请读者自行设计，但要合理。注意，该系统调用应该可以和 hide 系统调用共存，比如先用 hide_user_process 隐藏用户所有进程，然后用 hide 取消其中一个进程的隐藏。

（4）在/proc 目录下面创建一个文件/proc/hidden，该文件可读可写，对应一个全局变量 hidden_flag，当 hidden_flag 为 0 时，所有进程都无法隐藏，即便此前进程被 hide 系统调用要求隐藏。只有当 hidden_flag 为 1 时，此前通过 hide 调用要求被屏蔽的进程才隐藏起来。

（5）将进程隐藏功能变成一个内核可编译选项。

（6）设计一个内核模块，输出所有被隐藏的进程。

（7）在/proc 目录下创建一个文件/proc/hidden_process，该文件的内容包含所有被隐藏进程的 pid，各 pid 之间用空格分开。

10.1.2　背景知识介绍

命令 ps 和 top 都可以显示进程信息，在此简单介绍 ps，对 top 命令感兴趣的读者可自行查看 man 手册。ps 命令有很多选项，其中 ps aux 显示系统所有用户的所有进程的信息。下面是 ps aux 命令在笔者机器上产生的部分输出，其中 USER 是进程的拥有者，PID 是进程标识号，%CPU 是进程占用 CPU 的比例，%MEM 是进程占用系统存储器的比例，VSZ 是进程占

用虚拟内存的大小，RSS 是进程驻留集大小，TTY 是进程控制终端，STAT 是进程状态，START 是进程启动时间，TIME 是进程使用 CPU 的时间，COMMAND 是进程对应的程序名。

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.1	1988	656	?	S	21:13	0:00	init [3]
root	2	0.0	0.0	0	0	?	SN	21:13	0:00	[ksoftirqd/0]
xfs	1878	0.0	0.4	3648	1568	?	Ss	21:14	0:00	xfs -droppriv -daemon
root	1896	0.0	0.1	2156	452	?	Ss	21:14	0:00	/usr/sbin/atd
root	1922	0.0	0.3	3128	1176	?	Ss	21:14	0:00	cups-config-daemon
68	1932	0.1	0.8	4880	3216	?	Ss	21:14	0:01	hald
root	1964	0.0	0.3	2732	1236	?	Ss	21:14	0:00	login -- root
root	2023	0.0	0.3	5012	1532	tty1	Ss	21:14	0:00	-bash
root	2152	0.0	0.2	4364	900	tty1	R+	21:31	0:00	ps aux

Linux 操作系统下 ps 和 top 的实现都是利用 proc 文件系统提供的信息，程序包 procps 包含了这些利命令实现的源代码， 可从网址 http://procps.sourceforge.net/下载。下面列出 procps-3.2.7 包中一个简易 ps 实现的主函数，让读者以窥一斑。

```
1   int main(int argc, char *argv[]){
2       arg_parse(argc, argv);
3       if (!old_h_option){
4           const char *head;
5           switch(ps_format){
6           default: /* can't happen */
7           case 0:             head = "    PID TTY              TIME CMD"; break;
8           ......
9           case 'l'|0x80: head = "F     UID    PID    PPID PRI NI VSZ RSS WCHAN S    TTY
10            TIME COMMAND"; break;
11          }
12          printf("%s\n",head);
13      }
14      if (want_one_pid){
15          if (stat2proc(want_one_pid)) print_proc();
16          else exit(1);
17      }else{
18          struct dirent *ent;                 /* dirent handle */
19          DIR *dir;
20          int ouruid;
21          int found_a_proc;
22          found_a_proc = 0;
23          ouruid = getuid();
24          dir = opendir("/proc");
25          while(( ent = readdir(dir) )){
26              if (*ent->d_name<'0' || *ent->d_name>'9') continue;
```

```
27              if (!stat2proc(atoi(ent->d_name))) continue;
28              if (want_one_command){
29                  if (strcmp(want_one_command,P_cmd)) continue;
30              }else{
31                  if (!select_notty && P_tty_num==NO_TTY_VALUE) continue;
32                  if (!select_all && P_euid!=ouruid) continue;
33              }
34              found_a_proc++;
35              print_proc();
36          }
37          closedir(dir);
38          exit(!found_a_proc);
39      }
40      return 0;
41  }
```

对上面的代码简单解释如下：

第 2 行：分析 ps 命令行参数并设置相应的全局变量。

第 3～13 行：根据全局变量 old_h_option 和 ps_format 确定 ps 命令输出的标题栏。

第 14～16 行：处理 ps 指定进程号的情况，命令行方式是“ps –p pid”。全局变量 want_one_pid 放置了待处理的进程 id 号，stat2proc 函数从文件/proc/want_one_pid/stat 中读出进程的各种信息，放入一些全局变量中，这些变量由函数 print_proc 根据 ps_format 的要求选用并输出。

第 18～38 行：总的来说，就是使用 opendir 打开/proc 目录（第 24 行），循环读取/proc 的各项（第 25 行），然后过滤掉不满足条件的目录项，满足条件的则调用函数 print_proc 处理。

第 26 行：pid 号必然是数字打头。

第 27 行：如果目录项是 pid 号，则 stat2proc 处理成功。

第 28～29 行：处理命令行方式是“ps-C command”的形式，要求进程对应的可执行文件名是 command 才能输出。

第 31 行：如果 select_notty 为零，则跳过无控制终端的进程。

第 32 行：如果 select_all 为零，则跳过不属于当前用户的进程。

10.1.3 proc文件系统实现简介

前面已经提到，proc 不是一个磁盘文件系统，它的内容是动态生成的，在此我们介绍它的实现内幕。

如果内核编译包括了 proc 文件系统，则在内核启动时 proc_root_init 函数被调用，下面给出了该函数裁减后的代码：

```
1   void __init proc_root_init(void)
2   {
3       int err = proc_init_inodecache();
4       err = register_filesystem(&proc_fs_type);
```

```
5       proc_mnt = kern_mount(&proc_fs_type);
6       proc_misc_init();
7       proc_net = proc_mkdir("net", NULL);
8       proc_net_stat = proc_mkdir("net/stat", NULL);
9       proc_root_fs = proc_mkdir("fs", NULL);
10      proc_root_driver = proc_mkdir("driver", NULL);
11      proc_mkdir("fs/nfsd", NULL);
12      proc_bus = proc_mkdir("bus", NULL);
13      proc_sys_init();
14  }
15  static const struct file_operations proc_root_operations = {
16      . read              = generic_read_dir,
17      . readdir           = proc_root_readdir,
18  };
19  static const struct inode_operations proc_root_inode_operations = {
20      . lookup            = proc_root_lookup,
21      . getattr           = proc_root_getattr,
22  };
```

上面代码解释如下：

第 4 行：调用函数 register_filesystem 注册 proc 文件系统。

第 5 行：kern_mount 将创建文件系统的超级块，新建根结点的 proc_dir_entry 结构（描述见实验 5），为根结点建立 dentry 和 inode 结构。当根目录 inode 结点被初始化时，成员 i_op 和 i_fop 分别被设置为 proc_root_inode_operations 和 proc_root_operations。

第 6 行：创建根目录下的文件，如 loadavg ，uptime 等。

第 7～13 行：创建根目录下的子目录，如 net，fs 等。

对于 6～13 行代码，有两点需要特别说明。首先，proc 文件系统是树形结构，从根目录、子目录到叶子结点，它们对应内存中的一个 proc_dir_entry 结构，并没有磁盘上的对应结构。根目录下的文件如 loadavg 等也没有内容，只有当用户需要读取文件时才会动态生成。其次，此时 proc 文件系统还有大量的文件和目录没有生成，有些是在相关子系统的初始化时完成。但是，还有一些子树并不需要对应的 proc_dir_entry 结构，它们的内容完全是根据需要动态生成的，典型的例子就是/proc 目录下的进程信息。

当用户使用命令“ls /proc”或“ps aux”时，将会使用系统调用 readdir，根目录文件对象相应的 proc_root_readdir 函数将会被调用，代码如下：

```
1   static int proc_root_readdir(struct file * filp,
2                                void * dirent, filldir_t filldir)
3   {
4       unsigned int nr = filp->f_pos;
5       int ret;
6       if (nr < FIRST_PROCESS_ENTRY) {
7           int error = proc_readdir(filp, dirent, filldir);
8           if (error <= 0) {
9           unlock_kernel();
```

```
10                  return error;
11          }
12          filp->f_pos = FIRST_PROCESS_ENTRY;
13   }
14          ret = proc_pid_readdir(filp, dirent, filldir);
15          return ret;
16   }
```

第 4 行：nr 为待读目录项的位置。普通文件对象的内部偏移是以字节为单位计数，proc 文件系统根目录文件对象的偏移指针处理不同于此，具体分析见后。

第 6～13 行：从前面的 proc_root_init 函数已经看到，/proc 下面的目录和文件实际可分成两大类，一类是在 proc_root_init 等函数中创建的项，它们均有对应的 proc_dir_entry 结构，对它们来讲，每读取一个目录项后偏移加一；另一类是进程目录项，这些项是动态生成的。因为第一类的数量有限，目前远小于 FIRST_PROCESS_ENTRY（值为 256），所以当文件对象偏移小于 FIRST_PROCESS_ENTRY 时，proc_root_readdir 认为在读第一类对象，该工作由 proc_readdir 函数完成，因为这部分与任务无关，我们在此不做介绍。

第 14 行：当偏移大于 FIRST_PROCESS_ENTRY 或者第一类对象已经读完且用户缓冲区未满，调用 proc_pid_readdir 读取系统进程信息。函数代码如下：

```
1    #define TGID_OFFSET (FIRST_PROCESS_ENTRY + ARRAY_SIZE(proc_base_stuff))
2    int proc_pid_readdir(struct file * filp, void * dirent, filldir_t filldir)
3    {
4           unsigned int nr = filp->f_pos - FIRST_PROCESS_ENTRY;
5           struct task_struct *reaper=get_proc_task(filp->f_path.dentry->d_inode);
6           struct task_struct *task;
7           int tgid;
8           for (; nr < ARRAY_SIZE(proc_base_stuff); filp->f_pos++, nr++) {
9                   const struct pid_entry *p = &proc_base_stuff[nr];
10                  if (proc_base_fill_cache(filp, dirent, filldir, reaper, p) < 0)
11                          goto out;
12          }
13          tgid = filp->f_pos - TGID_OFFSET;
14          for (task = next_tgid(tgid);
15                  task;
16                  put_task_struct(task), task = next_tgid(tgid + 1)) {
17                  tgid = task->pid;
18                  filp->f_pos = tgid + TGID_OFFSET;
19                  if (proc_pid_fill_cache(filp, dirent, filldir, task, tgid) < 0) {
20                          put_task_struct(task);
21                          goto out;
22                  }
23          }
24   filp->f_pos = PID_MAX_LIMIT + TGID_OFFSET;
25   out:
26          put_task_struct(reaper);
```

```
27  out_no_task:
28          return 0;
29  }
```

上面的代码解释如下：

第 1 行：/proc 目录包含特殊的一项 self 进程目录项，self 实际上是当前读取/proc 的进程的链接。self 目录项在/proc 目录中的偏移为 FIRST_PROCESS_ENTRY。创建 self 目录项的信息在数组 proc_base_stuff 中，该数组大小为一。所以，/proc 目录下真正以进程号命名的子目录偏移是从 TGID_OFFSET 开始的，值为 FIRST_PROCESS_ENTRY+1。

第 8～12 行：如果当前偏移正好是 FIRST_PROCESS_ENTRY，则读取 self 项。

第 13～23：整体说来，就是扫描当前系统的进程，并以进程号为名生成目录项，然后将目录项信息填入用户缓冲区。

第 14 行：函数调用 next_tgid(tgid)返回线程组 id（task_struct 中的 tgid 成员）不小于参数 tgid 的进程描述符。请注意，/proc 目录下的进程号必须是线程组的组长。

第 18 行：进程目录项在/proc 目录中偏移为其 tgid 加上 TGID_OFFSET。之所以这样设计是为了实现的需要，如果/proc 目录的读取是通过多次 readdir 完成，下一次的读取便可以从上一次的 tgid 处开始。

第 19 行：将进程号（即是线程组号）生成的目录项填入用户缓冲区。

在实际应用，除了读取/proc 目录内容外，还经常读取/proc/pid 目录下进程的各种信息，做这些操作首先要使用 open 系统调用打开相应项。按照 4.3 节的介绍，第一次解析路径名/proc/pid 时，要用到根目录 inode 的 lookup 方法，即函数 proc_root_lookup，该函数在 dir 中查找是否存在 dentry 目录项，如果成功返回 NULL。该函数的代码如下：

```
static struct dentry *proc_root_lookup(struct inode * dir, struct dentry * dentry, struct nameidata*nd)
{
        if (!proc_lookup(dir, dentry, nd)) {
                return NULL;
        }
        return proc_pid_lookup(dir, dentry, nd);
}
```

可以看到，proc_root_lookup 的处理分为两步，首先，由 proc_lookup 查看参数 dentry 是否有对应的 proc_dir_entry 结构；其次，由 proc_pid_lookup 查看 dentry 是否为进程目录项，该函数是本次任务需要了解的。该函数源代码及解释如下：

```
1   struct dentry *proc_pid_lookup(struct inode *dir, struct dentry * dentry,
2    struct nameidata *nd)
3   {
4           struct dentry *result = ERR_PTR(-ENOENT);
5           struct task_struct *task;
6           unsigned tgid;
7           result = proc_base_lookup(dir, dentry);
8           if (!IS_ERR(result) || PTR_ERR(result) != -ENOENT)
```

```
9                   goto out;
10          tgid = name_to_int(dentry);
11          rcu_read_lock();
12          task = find_task_by_pid(tgid);
13          if (task)
14                  get_task_struct(task);
15          rcu_read_unlock();
16          if (!task)
17                   goto out;
18          result = proc_pid_instantiate(dir, dentry, task, NULL);
19          put_task_struct(task);
20  out:
21          return result;
22  }
```

第 7 行：判断是否是 self 目录项。

第 10 行：执行至此表明是非 self 项。把 dentry 的目录项名字转换成整数。

第 12 行：查找是否存在 tgid，存在则返回其相应的 task_struct 结构。

第 18 行：proc_pid_instantiate 为 task 分配一个 inode，并将该 inode 与 dentry 联系起来。

对 proc 文件系统实现的介绍到此为止，尽管还有许多方面没有涉及，但是完成本次任务已经足够了。

10.1.4 实验指南

在本次任务要求实现的多个子功能中，有些在前面的实验中已经详细介绍过，在此不再赘述。这里只详细介绍功能（1）、功能（5）和功能（7）的实现。

10.1.4.1 功能⑴的实现

该功能可以按下面的步骤进行：

（1）修改进程描述符结构 task_struct，该结构定义在 linux/sched.h 文件中，为其添加一个成员 cloak，用来记录进程隐藏与否。

（2）在进程创建时，把 task_struct 结构的成员 cloak 初始化为未隐藏。fork 系统调用的实现代码在文件 kernel/fork.c 中，具体实现的主要函数为 do_fork，请读者自己考虑合适的代码插入位置。这里有一个比较有意思的问题，就是子进程 cloak 初值是否应该继承父进程的原值，如果不继承，且父进程是隐藏的，我们可以通过暴露的子进程观察到父进程，这无疑是我们不期望的。如果采用继承的语义，则问题不仅仅涉及此处，比如 hide 系统调用，在隐藏一个进程的同时似乎也应隐藏该其子进程才能保持语义的一贯性。为了简化，我们采用子进程刚创建时不隐藏的策略，同时我们留一个问题给读者，如果采用进程隐藏意味着其子进程一定隐藏的语义，用户有没有办法发现隐藏的进程呢？

（3）添加 hide 系统调用。具体做法参见实验 6，这里有两处提醒读者，一是 sys_hide 的实现应该放在哪个文件中，也许文件 fs/proc/base.c 是合适的选择；另一个是关于 sys_hide 的实现，仅仅设置 task_struct 结构的 cloak 成员是不够的，参见第（6）步。

（4）修改 proc_pid_readdir 函数的代码，如果被读出的进程描述符的 cloak 成员值为 1，

则进程标识符不返还给用户缓冲区。

（5）修改 proc_pid_lookup 函数的代码，如果被找到的进程描述符的 cloak 成员值为 1，则不返回相关信息。

（6）解除已有的 dentry 项。前面已经提到，第（4）步是为了确保 readdir 系统调用读不到/proc 目录下被隐藏的进程号，而第（5）步为了确保 open 不能解析类似/proc/pid/xxx 之类的路径名，但是只有这两步是不够的，还必须考虑 VFS 层的缓冲问题，如果缓冲区里已经有相关数据，则操作会立即返回而不经过底层文件系统。许多文件系统的 readdir 实现都采用了页面缓冲，庆幸的是，proc_pid_readdir 没有这个问题，每次系统调用都会经过我们前面已经提到的执行路径。但是在 4.3 节中提到过，路径解析先访问 dcache，在 dache 中找不到相应项时才调用索引结点的 lookup 方法。因此 hide(pid, 1)的实现不仅要设置 cloak，而且要判断 pid 是否在 dcache 中存在，如果存在则要清除，否则下一次解析依然可能看见/proc/pid/。

清除 dcache 可采用 d_drop 函数，下面是来自 proc_flush_task 函数的示例代码，该函数在 fs/proc/base.c 中。

```
1   void proc_flush_task(struct task_struct *task)
2   {
3           struct dentry *dentry, *leader, *dir;
4           char buf[PROC_NUMBUF];
5           struct qstr name;
6           name.name = buf;
7           name.len = snprintf(buf, sizeof(buf), "%d", task->pid);
8           dentry = d_hash_and_lookup(proc_mnt->mnt_root, &name);
9           if (dentry) {
10                  shrink_dcache_parent(dentry);
11                  d_drop(dentry);
12                  dput(dentry);
13          }
14          ......
15  }
```

第 5 行：qstr 是用做路径解析的结构，解析之前先要填好路径名和长度。

第 6～7：将待查 pid 和长度分别填入 name.name 和 name.len。

第 8 行：根据 name 求得 hash 值，然后在 dcache 中查询是否有名字为 name.name 的目录项。

第 10 行：将查到的目录项从其父目录项的子目录项链表中删除。

第 11 行：将该目录项从 dcache 中移出。注意，此时如果还有另外的执行路径用到该目录项，该目录项依然存在，只是路径解析再也访问不到了。

读者可参考该代码完善 sys_hide 的实现。

10.1.4.2 功能（5）的实现

在此介绍如何添加内核编译选项。若没有现成的配置文件，在编译内核之前，首先要生成内核配置文件.config。当使用 make xconfig 或 make gconfig 等命令启动内核配置器时，配

置文件 arch/i386/Kconfig（假定工作平台是 i386）将被读取，该文件是主配置文件，可以包含内核树子目录下的配置文件，下面是该文件中的三行：

```
source "net/Kconfig"
source "drivers/Kconfig"
source "fs/Kconfig"
```

这三行表示读入 net，drivers 和 fs 三个子目录下的 Kconfig 文件。来看一下 fs/Kconfig 文件，我们感兴趣的内容如下:

```
1   menu "File systems"
2   ......
3   menu "Pseudo filesystems"
4
5   config PROC_FS
6           bool "/proc file system support" if EMBEDDED
7           default y
8           help
9             This is a virtual file system providing information about the status
10            of the system......
11
12  config PROC_KCORE
13          bool "/proc/kcore support" if !ARM
14          depends on PROC_FS && MMU
15  endmenu
16  ......
17  endmenu
```

可以看到，配置文件以嵌套菜单方式组织，层次清晰，可以参看 图 10-1。PROC_FS，PROC_KCORE等就是所谓的内核编译选项。首先要注意，内核编译选项在Kconfig文件中的名字省掉了前缀CONFIG，如PROC_FS在内核源代码中实际是CONFIG_PROC_FS。下面解释其中的几行：

第 6 和 13 行：内核编译选项可以是三种状态之一：y 为内核支持该选项；n 为内核摒除该功能；m 为该功能单独编译成模块。如果内核编译选项可以三者择一，用 tristate 来描述其取值范围。如果该选项不支持编译成模块，则用 bool 描述。PROC_FS 取值类型为 bool，表明 proc 文件系统要么编译进内核，要么内核不支持，不可能编译成模块。

第 7 行：default 表示内核编译选项的默认值，这里的 y 表示默认情况下内核加入 proc 文件系统支持。

第 8 行：帮助信息，当用户设置该选项值时显示。

第 14 行：PROC_KCORE 依赖于 PROC_FS 和 MMU 两个内核编译选项。

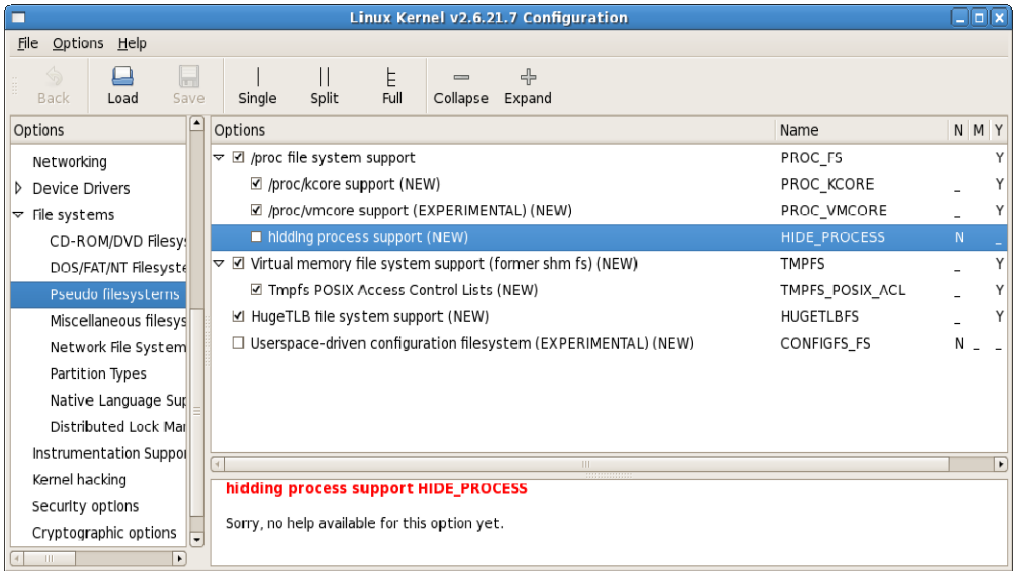

图 10-1　make gconfig（包含新内核编译选项）生成的界面

有了上面的知识，将隐藏进程功能变成内核编译实际是非常容易的，在 fs/Kconfig 文件中 PROC_KCORE 编译选项（即上面的 14 行）后添加如下几行：

```
config HIDE_PROCESS
    bool "hidding process support"
    depends on PROC_FS
    default n
```

图 10-1是作者为内核添加新选项后使用make gconfig生成的界面。

下面的事情是修改内核代码，该工作也很简单，主要添加条件编译，原来的代码必须在定义了 CONFIG_PROC_FS 的情况下才进行编译。以 linux/sched.h 为例，task_strcut 结构中成员 cloak 的处理方式如下：

```
#ifdef CONFIG_HIDE_PROCESS
    int cloak;
#endif
```

稍有麻烦的是 sys_hide 系统调用如何处理，虽然可以直接控制系统调用表，使是否包含 sys_stealth 依赖于 HIDE_PROCESS 选项，但这对用户的测试程序不友好。可采用下面的处理方式，系统调用表总是包含 sys_stealth,但其实现依赖于选项。

```
#ifdef CONFIG_HIDE_PROCESS
asmlinkage int sys_stealth(pid_t pid, int onoff)
{
    /*实现代码*/
}
#else
asmlinkage int sys_stealth(pid_t pid, int onoff)
{
```

```
        return -ENOSYS; /* ENOSYS 表示系统未实现该功能*/
    }
    #endif
```

10.1.4.3 功能（7）的实现

如何创建一个 proc 文件在实验 5 已经介绍过，在此介绍一个应该注意的问题。为了读 proc 文件，需要实现 read_proc_t 方法，其原型如下：

typedef int (read_proc_t)(char *page, char **start, off_t off, int count, int *eof, void *data);

在 9.4 节实验 5 中介绍的方法是假定文件内容不超过一个页面，文件的所有内容都往 page 中写，具体的偏移问题交给上一级的调用者处理，但是这种做法对于目前的问题存在缺陷。下面是出现问题的一个场景：假定/proc/hidden_process 内容的各个进程号是用空格分开，用户态程序对 hidden_process 文件的读是使用多次 read 系统调用完成的，如果 hidden_process 的内容如下：

1 2 3 4 5 6 7 8 123 124…

第 1 次 read 读出 1～5，文件偏移 offset＝9，如图 10-2 所示。

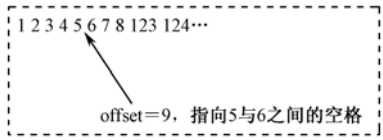

图 10-2 第 1 次读

在第 2 次 read 读之前，假定进程 2，3，4，5 均已释放，此时 hidden_process 的内容如下：

1 6 7 8 123 124…

假定此时 read 再读 offset=9 开始的内容，如图 10-3 所示。

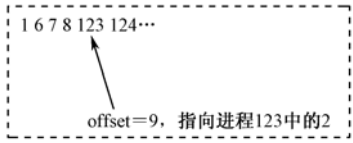

图 10-3 第 2 次读

可以看到，第 2 次 read 将读出“23 124”等内容，这样至少有两类错误，一是提供错误的“23”信息，二是系统的进程 6，7，8 被忽略掉了，以上这两类错误都是不能接受的。

解决上述问题的权宜之计是为用户态程序准备一个大的缓冲区，然后一次性将内容全部读出。但是当系统进程数目很多，一个页面放不下时，上面的方法便不奏效了，我们把该问题留给读者解决。

10.2 实验 8——字符设备驱动开发

Linux 内核代码中的驱动程序占了一半以上的代码量。一切设备皆文件，然而对文件的

操作如何影响到具体的设备呢？本次实验将通过一个实例和一个项目帮助大家解决上述问题，了解简单字符驱动开发的基本概念和流程，为从事具体的硬件设备驱动开发打下基础。

10.2.1 实验内容

实现一个字符设备驱动程序 chatdev.c，该设备并不驱动特定的硬件，它的功能是维护聊天会话信息。

实现一个用户态应用程序 chat.c，该程序可以向 chatdev 读/写消息，从而实现多个用户之间聊天的功能。

10.2.2 字符设备驱动开发介绍

干巴巴地讲述如何撰写一个字符设备驱动程序是件困难的事情，我们选择了一个简单的示例程序 chardev.c 来说明开发字符设备驱动要注意的问题。下面先列出程序代码：

```
1   #include <linux/init.h>
2   #include <linux/module.h>
3   #include <linux/cdev.h>
4   #include <linux/fs.h>
5   #include <asm/uaccess.h>
6   #include <linux/vmalloc.h>
7
8   #define MAX_DEVICES    10
9   #define MAJOR_NUM      101
10  #define MINOR_NUM      0
11  #define BUFFER_SIZE    1048576
12
13  struct _device_data
14  {
15      struct cdev         chardev;
16      unsigned char     * buffer;
17      int         npos;
18  }* mydata[MAX_DEVICES];
19  static ssize_t device_read(struct file* filp,
20                             char __user* buff, size_t len, loff_t* offset)
21  {
22      int nlen = len;
23      struct _device_data* pdb = filp->private_data;
24
25      if (nlen > pdb->npos - *offset)
26          nlen = pdb->npos - *offset;
27      if (copy_to_user(buff, (pdb->buffer) + *offset, nlen))
28       return -EFAULT;
29      *offset += nlen;
30      return nlen;
31  }
32  static ssize_t device_write(struct file* filp,
```

```
33                                  const char __user* buff, size_t len, loff_t* offset)
34  {
35      int nlen = len;
36      struct _device_data* pdb = filp->private_data;
37
38      if (nlen > BUFFER_SIZE - pdb->npos)
39          nlen = BUFFER_SIZE - pdb->npos;
40      if (nlen == 0)
41          return -ENOMEM;
42      if (copy_from_user(&pdb->buffer[pdb->npos], buff, nlen))
43          return -EFAULT;
44      pdb->npos += nlen;
45      return len;
46  }
47  static ssize_t device_open(struct inode* inode, struct file* filp)
48  {
49      int nminor = iminor(inode);
50
51      if (!mydata[nminor]->buffer)
52          mydata[nminor]->buffer = (unsigned char*)vmalloc(BUFFER_SIZE);
53      if (!mydata[nminor]->buffer)
54          return -ENOMEM;
55      filp->private_data = mydata[nminor];
56      if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
57          mydata[nminor]->npos = 0;
58      return 0;
59  }
60  static ssize_t device_release(struct inode* inode, struct file* filp)
61  {
62      return 0;
63  }
64  struct file_operations fops=
65  {
66      .owner = THIS_MODULE,
67      .read = device_read,
68      .write = device_write,
69      .open = device_open,
70      .release = device_release,
71  };
72  static int device_init(void)
73  {
74      int     i, ndev, ret;
75
76      printk(KERN_INFO "Loading " KBUILD_MODNAME " ...\n");
77      for (i = 0; i < MAX_DEVICES; ++i){
78          mydata[i] = kmalloc(sizeof(struct _device_data), GFP_KERNEL);
79          if (!mydata[i]){
80              printk(KERN_EMERG "Can't allocate memory to mydata\n");
```

```
81                  return -ENOMEM;
82              }
83              mydata[i]->buffer = NULL;
84              mydata[i]->npos = 0;
85              cdev_init(&mydata[i]->chardev, &fops);
86              mydata[i]->chardev.owner = THIS_MODULE;
87              ndev = MKDEV(MAJOR_NUM, MINOR_NUM + i);
88              ret = cdev_add(&mydata[i]->chardev, ndev, 1);
89              if (ret){
90                  printk(KERN_EMERG "Can't register device[%d]! \n", i, ret);
91                  return -1;
92              }
93          }
94          return 0;
95      }
96      static void device_exit(void)
97      {
98          int i;
99
100         printk(KERN_INFO "Unloading " KBUILD_MODNAME " ... \n");
101         for (i = 0; i < MAX_DEVICES; ++i){
102             cdev_del(&mydata[i]->chardev);
103             if (mydata[i]->buffer)
104                 vfree(mydata[i]->buffer);
105             kfree(mydata[i]);
106         }
107     }
108     module_exit(device_exit);
109     module_init(device_init);
110     MODULE_LICENSE("GPL");
```

10.2.2.1 测试字符设备

示例实现的字符设备实际上是内存文件，当掉电后设备内容即丢失。用户可以读/写该字符设备，操作方式和普通文件没有区别。

驱动程序以内核模块方式实现，按照实验 4 介绍的方法容易把 chardev.c 编译成目标文件 chardev.ko。然后可以写一个脚本程序 install.sh，内容如下：

```
#!/bin/sh
insmod ./chardev.ko
mknod /dev/chardev0 c 101 0
mknod /dev/chardev1 c 101 1
mknod /dev/chardev2 c 101 2
mknod /dev/chardev3 c 101 3
mknod /dev/chardev4 c 101 4
mknod /dev/chardev5 c 101 5
```

```
mknod /dev/chardev6 c 101 6
mknod /dev/chardev7 c 101 7
mknod /dev/chardev8 c 101 8
mknod /dev/chardev9 c 101 9

chmod 666 /dev/chardev*
```

执行该脚本程序后，用户便可以使用设备了。例如，使用下面的命令先将文件列表写入设备，然后显示设备的内容。

#ls -l > /dev/chardev0

#ls -a > /dev/chardev1

#cat /dev/chardev0

#cat /dev/chardev1

10.2.2.2 描述设备的数据结构

示例设备的描述结构是_device_data，如下所示：

```
struct _device_data
{
    struct cdev          chardev;
    unsigned char*       buffer;
    int                  npos;
}* mydata[MAX_DEVICES];
```

该驱动支持 MAX_DEVICES 个同类型的设备，主设备号为 MAJOR_NUM，次设备号从 0 到 MAX_DEVICES-1。chardev 成员是内嵌的 cdev 对象，描述见 10.2.2.4 节。buffer 成员是该设备存放数据的缓冲区，最大可达 BUFFER_SIZE（1MB）。npos 成员描述缓冲区已有的数据量。

10.2.2.3 设备号的操作

在 5.1 节中已经提到，每个设备都有主设备号和次设备号。当使用 mknod 命令生成设备文件时，该文件的 inode 便保存主设备号和次设备号的信息。内核访问该设备时，可通过下面两个函数取得主、次设备号，可参考示例第 49 行。

unsigned int iminor(struct inode *inode);

unsigned int imajor(struct inode *inode);

内核使用类型 dev_t 描述设备号，如果从其取得主设备号或次设备号，可以使用下面两个宏：

MAJOR(dev_t dev);

MINOR(dev_t dev);

有时候又要根据主设备号和次设备号生成 dev_t， 可使用下面的宏，见示例第 87 行。

MKDEV(int major, int minor);

10.2.2.4 字符设备的注册与注销

字符设备在内核中用结构struct cdev表示，字符设备在使用之前需要注册该结构[①]，该结构的头文件为<linux/cdev.h>。通常情况下cdev结构内嵌在描述设备的结构中，如示例第 15 行所示。这种情况下注册字符设备首先要使用函数cdev_init初始化cdev结构：

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

其中，参数 fops 为字符设备 cdev 对应的文件操作集。cdev 结构被初始化后，下一步调用函数 cdev_add 将设备注册到内核：

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

其中，参数 p 对应字符设备；dev 是该设备对应的第一个设备号；count 描述该设备对应的连续次设备号的个数，通常该值为 1。

以上两个函数的使用见示例第 85 行和 88 行，另外，第 86 行把字符设备的 owner 设为 THIS_MODULE 也是有必要的。

将字符设备从系统注销可使用函数 cdev_del 函数，见示例 102 行。

```
void cdev_del(struct cdev *dev);
```

10.2.2.5 文件操作集

对字符设备的操作是通过文件进行的，因此要实现结构 file_operations 的相关操作，示例 64～71 行包含了文件操作集中最重要的几个操作：打开、释放以及读/写，下面逐个讨论。

打开文件操作 device_open 主要用来做初始化工作，第 49 行根据 inode 得到该设备的次设备号。第 51 行根据次设备号检查该设备的缓冲区是否已经分配，如果未分配就使用 vmalloc 分配一块空间。第 55 行是极为关键的一步，将文件对象指针 filp 的成员 private_data 赋值为字符设备的数据结构，这样对文件的各种后续操作，如读/写等，都可以通过 private_data 得到描述字符设备的数据结构，进而可以操作设备，如示例 23 行和 36 行。

文件释放操作的任务一般是打开文件操作的逆过程，即释放打开操作中分配的资源，关闭设备。示例中 device_release 操作实现起来很简单，因为无须关闭设备。这里留一个问题给读者，device_open 中为设备分配了缓冲区，而释放缓冲区的操作却在模块卸载时在调用函数 device_exit 中进行（见第 104 行），这样做是否合理？

文件读操作的原型如下：

```
ssize_t device_read(struct file* filp, char __user* buff,
size_t len, loff_t* offset);
```

其中，filp 是文件对象指针；buff 是用户态缓冲区，用来接收读到的数据；len 是希望读取的数据量；offset 是用户访问文件的当前偏移。示例第 25～26 行保证 nlen 值不超过设备缓冲区中未读的数据量。第 27 行将设备缓冲区中未读的数据复制到用户态缓冲区，第 29 行调整偏移。

文件写操作的原型和读操作没有区别，只是操作方向改变而已。第 38 行检查设备缓冲区的空闲容量。第 42 行将用户态缓冲区中的数据复制到设备缓冲区中。第 44 行调整存入设备缓冲区的数据量。值得注意的是，该设备缓冲区的数据可以在多次 open ，close 操作之间

①为了兼容旧代码，2.6 内核依然支持使用 register_chrdev()函数注册字符设备，但在新代码中不推荐使用该函数。

依然存在，但是要使用追加数据的方式。下面给出了测试该功能的代码：

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void)
{
        int fd;
        int ch;

        if ((fd = open("/dev/chardev0", O_RDWR|O_APPEND)) == -1){
                fprintf(stderr, "can not open file chardev0\n");
                exit(0);
        }
        while ((ch = getchar()) != EOF){
                write(fd, &ch, 1);
        }
       close(fd);
       return 0;
}
```

10.2.2.6 同步

为叙述简单，示例程序中有些非常重要的问题没有考虑。以 device_open 为例，假定两个进程访问同一个设备，该设备的缓冲区尚未分配，两个进程同时执行第 51 行代码，均发现缓冲区为空，然后都去执行第 52 行代码，这将导致缓冲区被分配两次，而第一次分配的存储空间将丢失，这种情况我们称为 race condition，它正是系统设计所极力避免的。仔细观察不难发现，示例 device_read，device_write 的实现也不能阻止一些 race condition 的出现。

避免 race condition 出现的通用方法是，利用同步原语保护相关代码段。读者最熟知的同步原语莫过于信号量了，下面介绍 Linux 内核支持信号量操作的 API。

首先要包含相应的头文件<asm/semaphore.h>，信号量通常内嵌在需要保护的共享资源的描述结构中，以示例驱动为例，因为多个进程可能并发访问_device_data 结构，所以可以在该结构内添加如下成员：

```
struct semaphore sem;
```

下一步就是初始化信号量，可使用 sema_init 函数，参数 val 即是 sem 的初始值。

```
void sema_init(struct semaphore *sem, int val);
```

通常情况下，信号量是作为互斥量使用，这时可以选用下面两个 API，前者将信号量值初始化为 1，后者将信号量值初始化为 0。

```
void init_MUTEX(struct semaphore *sem);
```

void init_MUTEX_LOCKED(struct semaphore *sem);

内核对应信号量 P 操作有下面两个 API，down 和 down_interruptible。两者的区别在于如果操作处于阻塞状态，能否被用户中断。通常我们选用 down_interruptible。

void down(struct semaphore *sem);

int down_interruptible(struct semaphore *sem);

对应信号量 V 操作的内核 API 是 up。

void up(struct semaphore *sem);

10.2.3 字符设备chatdev的实现

大家司空见惯的聊天程序一般是用客户-服务器方式实现的，客户向服务器发送自己的聊天消息同时接收其他客户的消息，服务器端接收客户消息并把其发送给其他客户。利用字符设备实现聊天功能与利用客户-服务器来实现是有一定区别的。首先，字符设备无法向服务器一样，主动把消息发送给客户，它只能存储客户发给它的消息，这就意味着聊天程序 chat 必须亲自读取 chatdev 中保存的消息。其次，客户-服务器的并发处理很多是协议栈自动完成的，但是多个客户同时访问 chatdev，驱动必须亲自保证不发生 race condition。

下面讨论一个粗略的 chatdev 实现方案，读者自己可实现更加完美的方案。chatdev 要存储每个客户发来的消息，可采用下面的数据结构：

```
struct message {
    char text[MAX_LEN];          /*存放消息的缓冲区*/
    int text_length;             /*消息的实际长度*/
};
struct chatroom {
    struct semaphore sem;        /*互斥访问的信号量*/
    struct cdev chatdev;         /*内嵌的 cdev*/
    int index;                   /*  新消息的存放位置*/
    struct message msg[MAX_MSG];   /*  聊天室活跃消息容量  */
} chatroom[MAX_DEVICES];
```

每一个 chatdev 设备文件对应一个 chatroom 结构，每个 chatroom 可以容纳 MAX_MSG 条消息，假定 MAX_MSG 取值合理且客户总是在读取信息，我们可认为该容量总是足够的。消息数组 msg 循环使用，新的消息放在下标为 index 的位置，然后 index 加一。起初 index 为 0，index 逐渐增大到 MAX_MSG-1 后，下一条新消息的到来将使 index 重新变为 0，如此往复。

chatdev 的文件操作集类似于示例，列出如下：

```
struct file_operations chatdev_fops= {
    .owner = THIS_MODULE,
    .read = chatdev_read,
    .write = chatdev_write,
    .open = chatdev_open,
```

```
    .release = chatdev_release,
};
```

对于客户进程来说，要追踪进程自身在 chatroom 的读取位置。文件对象恰好有一个成员 f_pos 表示当前文件的偏移，把 f_pos 看成是消息数组 msg 的下标就可以解决前述问题。这样要涉及 chatdev_fops 的两个操作，chatdev_open 操作初始化文件对象的 f_pos 成员；而 chatdev_read 操作更新函数接口中的 offset 参数。

如果客户进程打开的设备文件对象偏移小于 chatroom.index，表明该进程还有消息未读取；如果偏移等于 chatroom.index，表明当前没有新的消息，此时 chatdev_read 应该立即返回而不应该阻塞。

10.2.4 聊天程序chat的实现

下面给出一个 chat 的实现方案，读者可以自行改进。

聊天进程的一个工作就是读取设备 chatdev，然后将消息显示在屏幕上。如果客户想发送消息，可以按【Ctrl+C】键触发信号处理程序，然后输入一条消息并将其写入 chatdev。下面给出 chat 程序的伪代码：

```
main(){
    设置 Ctrl+C 信号的处理程序 catchctrlc;
    while (1) {
        read(message, chatdev);
        display(message);
        sleep(1);/*睡眠 1 秒*/
    }
}
void catchctrlc(int signo)
{
    read(message, stdin); /*从键盘读入消息到 message*/
    if (str_equal(message, "exit"){
        清理现场;
        exit(0);
    }
    write(message, chatdev);
}
```

下面列出一个聊天场景：

```
#./chat
Enter your name: John
(按 Ctrl+C)
=>> Hi, Joe!
John: Hi, Joe!
(按 Ctrl+C)
=>> How are you?
```

Joe: Boy, am I starvin'!
Joe: Doing all that OS homework makes me hungry.
John: How are you?
(按 Ctrl+C)
=>> Me too，today I do nothing but play with my new chat program!
(按 Ctrl+C)
=>> exit
Goodbye.

10.3 实验 9——naive文件系统的设计与实现

本次实验的工作量较大。本次实验要求实现一个简单的文件系统 naive，我们把 naive 的开发分解成 10 个步骤，逐步完成这些步骤不仅可以减轻开发的难度，还可以充分认识文件操作 API 的实现流程，进一步熟悉 VFS 层的抽象语义。

10.3.1 实验内容

要求在 Linux 平台下实现对磁盘文件系统 naive 的支持，naive 文件系统的基本规格说明如下：

（1）块大小 512B。
（2）文件系统的大小不超过 2MB。
（3）文件/目录的最大长度不超过 4KB，文件/目录名不超过 30 个字符。
（4）超级块中必须有一成员项，用于放置学号。
（5）只需要支持普通文件和目录的基本操作，特殊类型的文件不需要考虑。

10.3.2 项目的准备工作及建议

本次任务需要内核支持环回（loopback）设备，对于常见的发行版，已经包含了环回设备，如 Fedora Core 5，就包含了/dev/loop0～/dev/loop7 等 8 个环回设备文件。Linux 文件系统的驱动支持以内核模块方式的加载，无须重新编译内核，这样可以节约不少开发时间。如果读者打算用自己编译的内核，编译时要选中支持环回设备的内核编译选项。

为完成该项目，可能需要阅读大量内核代码，建议多看看 minix 文件系统或 EXT2 文件系统的实现，我们推荐在 minixfs 或 EXT2 的基础上完成该任务，有些代码可以直接借用。但是必须指出，Linux 2.6 内核各个版本之间 API 可能会有变化，从而可能影响到 naivefs 的实现，一个具体的例子是 super_operations 中的 delete_inode 方法，在 2.6.14 版本以后的实现要求添加 truncate_inode_pages 功能。所以建议读者使用的开发平台内核版本是多少，就下载相应版本的内核进行开发。

对于直接实现有困难的读者，我们推荐另外一种路线。先把 naivefs 的要求放在一边，按照下面的具体实现步骤，裁减 minix 或 EXT2，必须保证每一步实现都是来自 minix 或 EXT2 实现的最小子集，当最后一步完成后，想必读者对文件系统的实现有了一定的了解，然后在此基础上再实现 naive。

10.3.3 实验指南

在下面的实现步骤中，为了突出主干的执行步骤和节省篇幅，里面列出的内核代码都经过了裁减，出错处理一般都省略掉了。

10.3.3.1 第 1 步——创建设备

尽管 Linux 支持像 procfs 和 ramfs 这样不需要物理磁盘的文件系统，naive 文件系统还是需要磁盘介质的。为了方便开发过程，可以利用 Linux 的环回设备创建一个伪磁盘设备，无须独立的新分区来存放 naive 文件系统。下面两条命令演示了如何处理：

#dd if=/dev/zero of=tmpfile bs=1k count=200

#losetup /dev/loop0 tmpfile

其中，命令 dd 创建一个大小为 200KB 的文件 tmpfile；losetup 将关联设备/dev/loop0 和普通文件 tmpfile，以后对设备的读/写操作就转换成对文件的读/写操作，也就是 tmpfile 被当成一个块设备了。如果想解除这种关联，可采用如下命令：

#losetup -d /dev/loop0

10.3.3.2 第 2 步——格式化分区

在建立环回设备后，接下来的工作就是对该设备进行格式化。Linux 平台下如果要把一个分区格式化为 EXT2 格式，可使用如下命令：

#mkfs -t ext2 /dev/loop0

其中，mkfs 实际是各种文件系统格式化工具的前端，mkfs -t type 实际上会调用 type 文件系统所提供的 mkfs.type 格式化工具。因此这里要实现应用于 naive 文件系统的 mkfs.naive 程序。

在讨论实现之前，先考虑 naive 文件系统的物理布局。naive 文件系统可以参照 EXT2 文件系统的布局，只是因为 naive 文件系统的容量有限，所以在设计上做不少简化。首先，没有必要分成多个组，整个文件系统就一个组，所以可以取消组描述符，组描述符中还有必要存在的成员项可以并入超级块，整体系统因而只有一个超级块，无须冗余备份。其次，数据块位图和索引结点位图依然占据一个块，假定块大小为 512B，则系统能支持的最大块数目为 512×8＝4096，能够支持的最大容量为 4096 块×512B/块=2097152B=2MB。同样地，索引结点位图占一个块意味着块大小为 512B 的情况下，系统能支持最多 4096 个文件。索引结点表占用块的数目取决于文件系统的索引结点数和索引结点本身的大小，索引结点的格式由读者自行设计，可在 ext2_inode 的基础上裁减，但是注意，该结构不能太大，否则索引结点将占用相当多的空间。

图 10-4 列出了 naive 分区的物理布局。

引 导 块	超 级 块	数据块位图	索引结点位图	索引结点表	数 据 块
	一个块	一个块	一个块	*n* 个块	*n* 个块

图 10-4 naive 分区的物理布局

由于 niave 文件系统比较简单，所以 mkfs.naive 的实现并不复杂，我们在此大致描述为如下过程：

（1）取得分区大小和打开设备，Linux 平台允许把设备像一个文件一样处理，代码如下：

```
fd = open(device_name,O_RDWR );
ioctl(fd, BLKGETSIZE, &size);
```

（2）为各种元数据开辟缓冲区，并初始化相应的成员项，因为超级块、数据块位图和索引结点位图都只占一个块，所以很容易处理。关键是索引结点表占多少个块，这取决于两个因素，一个是索引结点大小，另一个是索引结点数，索引结点数目肯定不超过设备的块数，如果假定文件平均至少两个块，则索引结点数为总块数的一半即可。

（3）处理根目录。刚格式化后的文件系统应该有一个根目录，所以应该从索引结点表中分配一个空闲索引结点（实际上，大部分文件系统都是指定某个索引结点为根索引结点），填充根索引结点各成员项，然后分配一个空数据块并在根索引结点中登记，然后在该数据块中写入两个目录项：.和..。

（4）将第（2）步中的缓冲区写入文件 fd。

对以上过程仍有疑惑的读者可以参考 mkfs.minix 或 mkfs.ext2 的实现。

10.3.3.3 第 3 步——定义并注册naive文件系统

在任何一个文件系统能够使用之前，都必须向内核注册。首先，需要填充 file_system_type 结构。该结构有多个成员，目前只需要填充 owner，name 和 fs_flags，name 是文件系统名，fs_flags 需要指定为 FS_REQUIRES_DEV，表示文件系统必须安装在物理磁盘上。

```
static struct file_system_type naive_fs_type = {
        .owner          = THIS_MODULE,
        .name           = "naive",
        .fs_flags       = FS_REQUIRES_DEV,
};
```

接下来，可以通过 register_filesystem 函数注册 naive 文件系统，注册过程应该放在模块加载时进行。当文件系统不再需要时，通过 unregister_filesystem 函数注销文件系统，下面是代码雏形：

```
static int __init init_naive_fs(void)
{
        int err;
        err = register_filesystem(&naive_fs_type);

        return err;
}
static void __exit exit_naive_fs(void)
{
        unregister_filesystem(&naive_fs_type);
}
module_init(init_naive_fs)
module_exit(exit_naive_fs)
```

读者补全这个模块后进行编译，假定生成的文件名叫 naivefs.ko。然后加载该模块，然后显示文件/proc/filesystems 的内容：

#insmod naivefs.ko

#cat /proc/filesystems

cat 命令的结果应该包括 naive 的一行，表示文件注册成功。然后使用如下命令注销文件系统：

#rmmod naivefs

再查看/proc/filesystems 便发现注销已经成功。

10.3.3.4 第 4 步——安装/卸载文件系统分区

注册文件系统后，便可以考虑安装前面第 2 步格式化的 loop 分区了。先看系统调用 mount 的执行过程：

sys_mount=>do_mount=>do_new_mount=>do_kern_mount

do_kern_mount 的大体流程如下：

```
struct vfsmount *
do_kern_mount(const char *fstype, int flags, const char *name, void *data)
{
    struct file_system_type *type = get_fs_type(fstype);
    struct super_block *sb = ERR_PTR(-ENOMEM);
    struct vfsmount *mnt;
    int error;
    char *secdata = NULL;
    ......
    mnt = alloc_vfsmnt(name);/*分配 vfsmount 结构*/
    ......
    sb = type->get_sb(type, flags, name, data);/*取得超级块*/
    ......
    /*初始化各成员项*/
    mnt->mnt_sb = sb;
    mnt->mnt_root = dget(sb->s_root);
    mnt->mnt_mountpoint = sb->s_root;
    mnt->mnt_parent = mnt;
    mnt->mnt_namespace = current->namespace;
    up_write(&sb->s_umount);
    put_filesystem(type);
    return mnt;
    ......
}
```

因此要实现 struct file_system_type 的 get_sb 方法，该方法得到一个超级块对象。参照其他磁盘文件系统的实现方法，可以利用内核辅助函数 get_sb_bdev。下面给出了具体的代码：

```
static struct super_block *naive_get_sb(struct file_system_type *fs_type,
```

```
                                        int flags, const char *dev_name, void *data)
{
        return get_sb_bdev(fs_type, flags, dev_name, data, naive_fill_super);
}
static struct file_system_type naive_fs_type = {
        .get_sb         = naive_get_sb,         /*安装时调用该方法*/
        .kill_sb        =   kill_block_super,   /*卸载时调用该方法*/
        ......
};
```

get_sb_bdev 会分配一个超级块并初始化其中的若干成员项，但是还需要 naive 文件系统实现一个 naive_fill_super 函数来读出磁盘上的超级块信息。该函数的原型如下：

```
static int    (*fill_super)(struct super_block *s, void *data, int silent)
```

磁盘文件系统 fill_super 函数的实现包括下面 4 个相似步骤：

```
struct buffer_head *bh;
    struct inode *root_inode;
    bh = sb_read(s, NAIVE_SUPER_BLOCK);          /*①*/
    s->s_op = &naive_sops;                       /*②*/
    root_inode = iget(s, NAIVE_ROOT_INO);        /*③*/
    s->s_root = d_alloc_root(root_inode);        /*④*/
```

第①步内核函数 sb_read 从磁盘上读出超级块的内容；第②步设置超级块对象的操作集；第③步内核函数 iget 利用超级块对象操作集从磁盘上读出根结点信息；第④步为根结点生成相应的 dentry 项。因此实现 naive_fill_super 之前我们还要先实现超级块对象操作集 naive_sops。

```
static struct super_operations naive_sops = {
        .alloc_inode        = /* naive_alloc_inode 或 NULL */,
        .destroy_inode      = /* naive_destroy_inode 或 NULL */,
        .read_inode         = naive_read_inode,
        .put_super          = naive_put_super,
};
```

为实现从磁盘上读 inode，必须实现 naive_read_inode 方法。在读 inode 之前必须先分配一个 inode 存放信息，Linux 有两种做法。一种是利用通用的 inode，这种情况下 alloc_inode 和 destroy_inode 都无须实现。许多物理文件系统采用另一种做法，文件系统特定的 inode 结构内嵌通用 inode，用 slab 机制来实现该类型对象的分配。以 EXT2 为例，该结构为 ext2_inode_info。

```
struct ext2_inode_info {
    __le32      i_data[15];
    __u32       i_flags;
    .......
    struct inode        vfs_inode;/*内嵌的通用 inode*/
```

```
};
static struct super_operations ext2_sops = {
        .alloc_inode  = ext2_alloc_inode,
        .destroy_inode       = ext2_destroy_inode,
};
static kmem_cache_t * ext2_inode_cachep;
static struct inode *ext2_alloc_inode(struct super_block *sb)
{
        struct ext2_inode_info *ei;
        ei = (struct ext2_inode_info *)kmem_cache_alloc(ext2_inode_cachep, SLAB_KERNEL);
        ......
        return &ei->vfs_inode;
}
static void ext2_destroy_inode(struct inode *inode)
{
        kmem_cache_free(ext2_inode_cachep, EXT2_I(inode));
}
```

下面介绍 read_inode 方法。该方法的原型如下：

void (*read_inode) (struct inode * inode);

其中，inode 的 i_ino 成员（inode 号）已经设置，read_inode 根据 i_ino 从磁盘上读入信息填充 inode。下面通过 ext2_read_inode 的实现来说明，应该注意的地方我们添加了注释。

```
void ext2_read_inode (struct inode * inode)
{
        struct ext2_inode_info *ei = EXT2_I(inode);
        ino_t ino = inode->i_ino;
        struct buffer_head * bh;

        /*ext2_get_inode 从磁盘上读出 inode 放入 raw_inode*/
        struct ext2_inode * raw_inode = ext2_get_inode(inode->i_sb, ino, &bh);
```

利用 raw_inode 初始化 inode 和 ei；

```
/*根据文件的类型设置 i_op、i_fop 和 i_mapping->a_ops 三个操作集，
*目前我们设置为空即可。
*/
        if (S_ISREG(inode->i_mode)) { /*如果是普通文件*/
                inode->i_op = &ext2_file_inode_operations;
                inode->i_fop = &ext2_file_operations;
                if (test_opt(inode->i_sb, NOBH))
                        inode->i_mapping->a_ops = &ext2_nobh_aops;
                else
                        inode->i_mapping->a_ops = &ext2_aops;
        } else if (S_ISDIR(inode->i_mode)) {    /*如果是目录*/
                inode->i_op = &ext2_dir_inode_operations;
```

```
                inode->i_fop = &ext2_dir_operations;
                if (test_opt(inode->i_sb, NOBH))
                        inode->i_mapping->a_ops = &ext2_nobh_aops;
                else
                        inode->i_mapping->a_ops = &ext2_aops;
        }
        ......
}
```

小结：文件系统能被安装，需要实现 naive_get_sb，naive_fill_super，naive_sops 中的 read_inode 等方法。

当卸载一个文件系统时，需要调用 file_system_type 对象的 kill_sb 方法，幸运的是，内核提供了函数 kill_block_super 可完成大部分功能，我们只需要实现 super_operations 集合中的 put_super 方法供 kill_block_super 调用即可。put_super 的实现较为简单，基本的操作有更新超级块、释放 naive_fill_super 申请资源等。

完成代码后编译新完成的模块，使用 insmod 加载模块。然后执行下面命令安装设备/dev/loop0。

#mkdir /mnt/naive

#mount -t naive /dev/loop0 /mnt/naive

执行下面命令可看到安装分区的信息:

#cat /proc/mounts

然后再用下面两条命令测试 umount 的效果。

#umount /dev/loop0

#cat /proc/mounts

除了使用前面的命令外,最好不要试图对/mnt/naive 进行任何其他操作(如 ls, mkdir 等)，因为文件系统还未开发完全。但是，假设读者好奇心强，使用了 ls 命令，产生的结果你能通过读内核代码的方式得到确认吗？

10.3.3.5 第 5 步——显示根目录的内容

现在要对 naive 文件系统添加最简单的目录功能，我们的目标是读出 naive 文件系统的根目录，此时根目录下只有.和..两项。首先，我们来看支持目录显示需要添加的功能。要实现读目录必须支持 file_operations 和 inode_operations 两个操作集合中的相应方法，由第 4 步中对 ext2_read_inode 函数的注释可以看到，普遍文件和目录的操作集合通常是不同的，这里暂时只需要支持目录对象的 file_operations 和 inode_operations。假设要显示的目录包含打开目录和读取目录两步操作，下面依次讨论内核对这两方面的支持。

内核打开目录也是用 sys_open 系统调用，4.3 节已经提到过实现过程包括解析路径，分配一个文件对象，然后返回文件描述符。解析路径需要我们提供 inode_operations 的 lookup 方法。因为目前我们访问的只是 naivefs 文件系统的根目录，其相对应的 inode 已被读入，根本无须解析，所以 naive_lookup 暂时只是一个空壳就可以满足要求，下面列出了相关代码：

```
static struct dentry *naive_lookup(struct inode * dir, struct dentry *dentry, (struct nameidata *nd)
```

```
{
        return NULL;
}
struct inode_operations naive_dir_inode_operations = {
        .lookup         = naive_lookup,
};
```

顺便提一下，实现一个空壳的 naive_lookup 是有必要的，否则打开目录函数 opendir 只会返回一个 ENOTDIR 错误（名字不是一个目录）。对此感兴趣的读者可阅读内核源代码，下面列出的位于函数 link_path_walk（fs/namei.c）中的代码可以说明这一现象。

```
if (lookup_flags & LOOKUP_DIRECTORY) {
        err = -ENOTDIR;
        if (!inode->i_op || !inode->i_op->lookup)
           break;
}
```

读取目录需要实现 file_operations 中的 readdir 方法，内核对 readdir 的调用过程如下：

sys_getdents（或 sys_getdents64）=>vfs_readdir=>file->f_op->readdir

readdir 原型如下：

int (*readdir) (struct file * filp, void * dirent, filldir_t filldir);

readdir 方法的基本功能是，从文件对象指针 filp 得到文件操作的当前位置（f_pos 成员），并开始读取目录项，然后利用填充函数 filldir 将读到的内容写入缓冲区 dirent。我们以 ext2_readdir 的代码（这里给出的是裁减版本）为例来说明实现应注意的一些问题。

```
static int ext2_readdir (struct file * filp, void * dirent, filldir_t filldir)
{
        loff_t pos = filp->f_pos;
        struct inode *inode = filp->f_dentry->d_inode;
        struct super_block *sb = inode->i_sb;
        unsigned int offset = pos & ~PAGE_CACHE_MASK;        /*pos 在页内的偏移*/
        unsigned long n = pos >> PAGE_CACHE_SHIFT;           /*pos 所在页号*/
        unsigned long npages = dir_pages(inode);             /*inode 的页面数*/
        unsigned char *types = NULL;
        int ret;

        /*已经到了尾部*/
        if (pos > inode->i_size - EXT2_DIR_REC_LEN(1))
                goto success;

        if (EXT2_HAS_INCOMPAT_FEATURE(sb, EXT2_FEATURE_INCOMPAT_FILETYPE))
                types = ext2_filetype_table;


        for ( ; n < npages; n++, offset = 0) {/*每次一页*/
```

```
                char *kaddr, *limit;
                ext2_dirent *de;
                struct page *page = ext2_get_page(inode, n);/*读取页面*/

                kaddr = page_address(page);
                de = (ext2_dirent *)(kaddr+offset);
                limit = kaddr + ext2_last_byte(inode, n) - EXT2_DIR_REC_LEN(1);
                for ( ;(char*)de <= limit; de = ext2_next_entry(de)) {/*每次一项*/
                        if (de->inode) {
                                int over;
                                unsigned char d_type = DT_UNKNOWN;

                                if (types && de->file_type < EXT2_FT_MAX)
                                        d_type = types[de->file_type];

                                offset = (char *)de - kaddr;

                                /*将目录项信息写入 dirent,如果成功，返回 0；如果失败，返回负值
                                （通常是因为缓冲区已满）*/
                                over = filldir(dirent, de->name, de->name_len,
                                                (n<<PAGE_CACHE_SHIFT) | offset,
                                                le32_to_cpu(de->inode), d_type);
                                if (over) {
                                        ext2_put_page(page);
                                        goto success;
                                }
                        }
                        filp->f_pos += le16_to_cpu(de->rec_len);/*调整偏移*/
                }
                ext2_put_page(page);
        }

success:
        ret = 0;
done:
        filp->f_version = inode->i_version;
        return ret;
}
static struct page * ext2_get_page(struct inode *dir, unsigned long n)
{
        struct address_space *mapping = dir->i_mapping;
        struct page *page = read_cache_page(mapping, n,
                                (filler_t*)mapping->a_ops->readpage, NULL);
        ......
}
```

Linux读取目录也利用page cache机制，以ext2_get_page函数为例，读取页面可以利用内

核函数read_cache_page，但是文件系统需要实现address_space_operations操作集的readpage方法。内核提供了辅助函数block_read_full_page便于实现readpage[①]。所以我们实现naive_readpage有如下形式：

```
static int naive_readpage(struct file *file, struct page *page)
{
    return block_read_full_page(page,naive_get_block);
}
static struct address_space_operations naive_aops = {
    .readpage = naive_readpage,
};
```

因此我们只要实现一个 naive_get_block 即可，其原型如下：

```
int   naive_get_block(struct inode *inode, sector_t iblock,
                      struct buffer_head *bh_result, int create);
```

函数从 inode 中读块号为 iblock（注意相对于文件的块号，而不是相对于设备的块号）的内容放入 bh_result 中。create 为 0 表示读，为 1 表示写。

完成上面的所有工作后，别忘了在 naive_read_inode 中添上如下的代码：

```
if (S_ISREG(inode->i_mode)) {
    inode->i_op = NULL;
    inode->i_fop = NULL;
    inode->i_mapping->a_ops = NULL;
} else if (S_ISDIR(inode->i_mode)) {
    inode->i_op = &naive_dir_inode_operations;
    inode->i_fop = &naive_dir_operations;
    inode->i_mapping->a_ops = &naive_aops;
}
```

生成新的模块后将其加载，然后进行测试。为了能够清晰地显示我们工作的结果，在安装/dev/loop0 之前，先在/mnt/naive 创建一个 origin 目录。执行下面的命令：

#mkdir /mnt/naive/origin

#ls /mnt/naive -a

ls 命令的结果应该显示三项——origin,.和..。现在安装/dev/loop0，执行下面的命令：

#mount -t naive /dev/loop0 /mnt/naive

#ls /mnt/naive –a

ls 命令的结果应该只有两项——.和..，即 naivefs 文件系统根目录的内容。

10.3.3.6　第 6 步——在根目录下创建内容为空的文件

本步骤的目标是在根目录下创建零长度的文件，然后通过 ls 命令观察新建项。用户态库

① ext2_readpage 采用了 mpage_readpage 来提高性能，这里我们选用简单的方案。

函数 creat 或 open 文件时，使用 O_CREAT 标志均可以创建新文件，相对应的内核系统调用函数为 sys_open。创建新文件的执行路径如下：

sys_open=>filp_open=>open_namei=>vfs_create=>dir->i_op->create

可以看到，需要实现针对目录的 inode_operations 操作集中的 create 方法。所以 naive_dir_inode_operations 应有内容如下：

```
struct inode_operations naive_dir_inode_operations = {
        .create         = naive_create, /*本步骤将要实现*/
        .lookup         = naive_lookup,  /*已实现为一个空壳*/
};
```

create 方法原型如下：

int create (struct inode * dir, struct dentry * dentry, int mode, struct nameidata *nd);

其中，dir 为目录的 inode 指针，dentry 为指向新建文件的目录项，mode 为新建文件的模式。

我们依然以 EXT2 的实现来说明主要步骤。

```
static int ext2_create (struct inode * dir, struct dentry * dentry, int mode, struct nameidata *nd)
{
    /*ext2_new_inode()分配一个 inode 结点并初始化各成员*/
    struct inode * inode = ext2_new_inode (dir, mode);
    int err = PTR_ERR(inode);
    if (!IS_ERR(inode)) {
        /*初始化 inode 的三个操作集*/
        inode->i_op = &ext2_file_inode_operations;
        inode->i_fop = &ext2_file_operations;
        if (test_opt(inode->i_sb, NOBH))
            inode->i_mapping->a_ops = &ext2_nobh_aops;
    else
            inode->i_mapping->a_ops = &ext2_aops;
        mark_inode_dirty(inode); /*标记 inode 为脏，但还未写入磁盘*/
        err = ext2_add_nondir(dentry, inode);/*添加目录项*/
    }
    return err;
}
```

ext2_add_nondir 将新文件的目录项添加到 dir 目录的数据块中，这涉及对页面缓冲的写操作，需要实现 address_space_operations 操作集中了三个方法：writepage，prepare_write 和 commit_write。注意，大部分文件系统的实现都使用了内核提供的辅助函数 block_write_full_page，block_prepare_write 和 generic_commit_write，使这三者非常易于实现。

完成上面的所有工作并生成新的模块后将其加载，然后安装 naive 文件系统到目录 /mnt/naive,开始进行测试。我们输入下面两条命令：

#touch /mnt/naive/zero_length_file

#ls -a /mnt/naive

touch 命令可以生成一个大小为零的文件。如我们所期待的，相比前一步，ls 的显示多出一项 zero_length_file，工作似乎完成了，果真如此吗？请依次执行下面三条命令：

#umount /mnt/naive

#mount -t naive /dev/loop0 /mnt/naive

#ls -a /mnt/naive

这一次 ls 命令的执行结果看不到 zero_length_file，导致这个错误出现的原因是，我们没有提供 super_operations 操作集的 write_inode 方法，根结点的 inode 数据没有被刷新到磁盘，导致 inode 的大小没有改变，也就意味着目录项没有增加，所以 umount 后再次 mount 看不到 zero_length_file，同时要注意 zero_length_file 的 inode 数据也没有写到磁盘，所以此时文件系统是不一致的。

我们以 ext2_write_inode 为例，说明 write_inode 实现的主要思路。ext2_write_inode 调用 ext2_update_inode 函数，该函数读出位于内存中 inode 在磁盘上对应的索引结点 raw_inode，然后刷新该结点，标记它所在的缓冲区为脏，因为系统会在合适的时刻刷新 buffer cache，所以 inode 数据能得到更新。

```
static int ext2_update_inode(struct inode * inode, int do_sync)
{
    struct super_block *sb = inode->i_sb;              /*sb 为 inode 所在超级块*/
    ino_t ino = inode->i_ino;                          /*inode 在磁盘上的 inode 号*/
    struct super_block *sb = inode->i_sb;
    struct buffer_head * bh;                           /*buffer cache 的管理结构*/

    /* ext2_get_inode 分配一个 buffer cache，用 bh 管理，根据 ino 读出 inode 在磁盘上对应的
     物理索引结点 raw_inode，放在 bh 所指向的缓冲*/
    struct ext2_inode * raw_inode = ext2_get_inode(sb, ino, &bh);
    用 inode 各成员对 raw_inode 各成员赋值；
    mark_buffer_dirty(bh);                                  /*标记脏*/
    brelse (bh);
}
```

完成 naive_write_inode 后，重复前面的测试过程，应该可以得到预期的结果了。

10.3.3.7 第 7 步——写文件和读文件

要支持常规文件的读/写操作，首先，要实现文件 inode 的两个操作集 inode_operations 和 file_operations；其次，要支持解析文件名，必须实现 naive_dir_inode_operations 中的 naive_lookup 方法,而不是像前面那样只提供一个空壳。下面依次讨论它们。

操作集 inode_operations 中大部分方法是针对目录类型，所以文件 inode 的该操作集依然保持为空，不做任何工作。而对于操作集 file_operations，read 和 write 两个方法是必须实现的，大部分文件系统都利用内核提供的通用函数，如下所示：

```
struct file_operations naive_file_operations = {
    .read       = generic_file_read,
```

```
    .write      = generic_file_write,
};
```

generic_file_read/write 会利用操作集 address_space_operations 的 readpage 和 writepage 方法，这两者我们已经实现了。接下来要修改 naive_read_inode 的相关部分，利用下面的代码设置操作集：

```
if (S_ISREG(inode->i_mode))
 {
    inode->i_op = NULL;
    inode->i_fop = naive_file_operations;
    inode->i_mapping->a_ops = NULL;
}
```

下面讨论 inode_operations 中 lookup 方法的实现。按惯例用 ext2_lookup 来说明主要步骤，参数 dir 是被查找目录的索引结点，参数 dentry 是被查找的目录项。

ext2_inode_by_name 函数根据 dentry->d_name 找出该目录项所对应的索引结点号 ino，然后 ino 作为参数调用 iget 读出相应的 inode，最后将 dentry 和 inode 关联起来。

```
static struct dentry *ext2_lookup(struct inode * dir, struct dentry *dentry, struct nameidata *nd)
{
    struct inode * inode;
    ino_t ino;
    ino = ext2_inode_by_name(dir, dentry);
    inode = NULL;
    if (ino) {
        inode = iget(dir->i_sb, ino);
    }
    if (inode)
        return d_splice_alias(inode, dentry);
    d_add(dentry, inode);
    return NULL;
}
```

完成本步骤后，假定当前目录为/mnt/naive,可以使用如下命令进程初步测试：

#echo abcde > a.txt

#cat a.txt

#cp a.txt b.txt

10.3.3.8 第 8 步——删除文件

删除文件要用到的系统调用是 sys_unlink，需要分两个步骤完成。

第 1 步，删除文件对应的目录项，其内核执行路径是 sys_unlink=>vfs_unlink=>dir->i_op->unlink，所以要实现 naive_dir_inode_operations 操作集中的 unlink 方法。我们以

ext2_unlink 为例来说明其主要步骤，参数 dentry 是待删除的目录项，参数 dir 为目录的 inode 结点。前面已经提到过，读/写目录都是通过页面缓冲完成的，ext2_find_entry 以页为单位逐次读取目录内容，如果目录项在页面中，则将该页面位置放入变量 page 中，目录项在 page 中的位置记录到变量 de 中。ext2_delete_entry 删除页 page 的 de 项，然后将页 page 写回磁盘。

```
static int ext2_unlink(struct inode * dir, struct dentry *dentry)
{
        struct inode * inode = dentry->d_inode;
        struct ext2_dir_entry_2 * de;
        struct page * page;
        int err = -ENOENT;
        de = ext2_find_entry (dir, dentry, &page);
        err = ext2_delete_entry (de, page);
        inode->i_ctime = dir->i_ctime;
        ext2_dec_count(inode); /*inode 的链接数减一*/
}
```

第 2 步，删除文件对应的 inode 和相应的数据块。传统的 UNIX 文件系统支持硬链接（naive 文件系统并不要求支持），一个 inode 可能有多个别名，只有所有的名字都删除后（即 inode 的链接数变为零）才能删除 inode。其内核执行路径是 sys_unlink=>iput=>iput_final=>generic_drop_inode=>generic_delete_inode=> inode->i_sb->s_op-> delete_inode。因此要实现超级块操作集 super_operations 中的 delete_inode 方法。下面照旧以 EXT2 文件系统的实现 ext2_delete_inode 来说明。首先，ext2_truncate 要清除 inode 在 page cache 和 buffer cache 中的数据，然后释放相应的数据块；然后，ext2_free_inode 释放磁盘上 inode 结点。

```
void ext2_delete_inode (struct inode * inode)
{
        ......
        inode->i_size = 0;
        if (inode->i_blocks)
                ext2_truncate (inode);
        ext2_free_inode (inode);
}
```

添加完上述功能后，可以使用 rm 命令来测试新功能。

10.3.3.9 第 9 步——创建目录

创建目录的系统调用是 sys_mkdir，其内核执行路径为 sys_mkdir => vfs_mkdir=> dir->i_op->mkdir，因此需要实现 naive_dir_inode_operations 操作集中的 mkdir 方法。以 ext2_mkdir 为例，从下面的代码可以看到，ext2_mkdir 的实现类似于 ext2_create，这是很好理解的，但是新建目录要比新建文件复杂一点，因为新建的目录不为空，它要新建两项.和..，这正是 ext2_make_empty 函数的工作。

```
static int ext2_mkdir(struct inode * dir, struct dentry * dentry, int mode)
{
	struct inode * inode;
	int err = -EMLINK;
	ext2_inc_count(dir);
	inode = ext2_new_inode (dir, S_IFDIR | mode);
	inode->i_op = &ext2_dir_inode_operations;
	inode->i_fop = &ext2_dir_operations;
	if (test_opt(inode->i_sb, NOBH))
		inode->i_mapping->a_ops = &ext2_nobh_aops;
	else
		inode->i_mapping->a_ops = &ext2_aops;
	ext2_inc_count(inode);
	err = ext2_make_empty(inode, dir);
	/*ext2_create 中的函数 ext2_add_nondir 就是 ext2_add_link 的封装*/
	err = ext2_add_link(dentry, inode);
	d_instantiate(dentry, inode);
}
```

完成上述功能后，可以使用 mkdir 命令来测试新功能。

10.3.3.10　第 10 步——删除目录

删除目录的系统调用是 sys_rmdir，其内核执行路径为 sys_rmdir => vfs_rmdir=> dir->i_op->rmdir，因此需要实现 naive_dir_inode_operations 操作集中的 rmdir 方法。我们以下面的 ext2_rmdir 代码为例。可以看到，先调用 ext2_empty_dir 判断目录是否为空，如果不为空，则调用删除文件的实现 ext2_unlink。

```
static int ext2_rmdir (struct inode * dir, struct dentry *dentry)
{
	struct inode * inode = dentry->d_inode;
	int err = -ENOTEMPTY;
	if (ext2_empty_dir(inode)) {
		err = ext2_unlink(dir, dentry);
		if (!err) {
			inode->i_size = 0;
			ext2_dec_count(inode);
			ext2_dec_count(dir);
		}
	}
	return err;
}
```

完成上述功能后，可以使用 rmdir 命令来测试新功能。

至此，一个具备简单功能的文件系统就完成了，读者可根据自己的兴趣添加新的功能，

如支持链接和文件重命名等。

10.4 实验 10——块设备驱动开发

块设备种类多，使用广泛，其驱动程序的开发也比字符设备复杂。通过本实验，读者要开发一个实际块设备（U 盘）的驱动程序，由此更深入地掌握块设备驱动程序的开发方法。Linux 下已经有一个通用的 U 盘驱动程序 usb-storage.ko，其源程序放在目录 drivers\usb\storage 下（相对于内核源码根目录）。但这个驱动的实现相当复杂，本实验希望开发一个相对简单的 U 盘驱动程序，不求高性能，只求结构明朗、清晰易懂，主要是让读者掌握一个实际块设备的驱动方式，从而加深理解。

事实上，本实验开发的驱动程序应该能够适用于所有基于 Bulkonly 传输协议的 USB 大容量存储设备（USB Mass Storage），如 USB 移动硬盘和 USB 外置光驱，而 USB 闪存盘（U 盘）只是其中的一种。由于 USB 大容量存储设备具有容量大、速度快、连接灵活、即插即用、总线供电等优点，它们已得到广泛使用，掌握这类设备驱动程序的开发技术无疑具有很强的实用性。

10.4.1 实验内容

编写一个 U 盘驱动程序 simple-udisk，只要求能够驱动某个型号的 U 盘，能够支持 U 盘的常规操作，如命令 hexdump，mke2fs 和 mount 等。同时要求在系统内核日志中显示出 U 盘的容量。若有余力，可增加多分区支持功能。

10.4.2 实验基础和思路

在参考文献[2]中，讲解了如何编写一个 Ramdisk 块设备驱动程序，称为 sbull；同时也讲解了如何编写一个 USB 设备驱动程序，并以 Linux 源代码中的 usb-skeleton.c 为例。虽然前者驱动的并不是一个实际的块设备，且后者又只是针对 USB 字符设备，但是它们提供了一个不错的基础，通过合并我们就能基本得到一个支持 USB 块设备的驱动程序。之所以说基本得到，是因为合并后只是有了块设备、USB 设备的驱动支持框架，但还缺一样：对 U 盘（USB 块设备）的实际访问操作。USB 块设备的访问方法与 USB 字符设备区别很大，有一套复杂的协议[10]。把这样一套协议研究清楚，将花费大量的时间，也远离了我们驱动程序开发的核心。这是一大难点，为此我们专门编写了一个 U 盘访问函数（my_Bulk_transport），以减轻工作量。10.4.3 节将对该函数的使用方法和工作过程进行专门讲解。

简言之，合并 sbull 和 usb-skeleton 这两个参考驱动程序，以构造整体框架，调用帮助函数 my_Bulk_transport 以访问 U 盘，从而打造一个简洁的 U 盘驱动程序。本节接下来介绍这两个参考驱动程序 sbull 和 usb-skeleton，着重讲解其工作原理及合并的关键环节。

10.4.2.1 参考驱动程序 1——块设备驱动程序sbull

参考文献[2]中第 16 章详细讲解了一个 Ramdisk 设备的驱动程序 sbull。所谓 Ramdisk，是指使用计算机内存作为存储介质的盘。一旦加载该驱动，就会从内存中划分出一些空间，虚拟出几个磁盘，并且可以对这个虚拟磁盘进行格式化、文件复制和查看等操作。

首先我们从网址 http://www.oreilly.com/catalog/linuxdrive3 下载 sbull 的源代码，包括两个

文件 sbull.c 和 Makefile，将这两个文件复制到 Linux 的某个目录下，并且在终端窗口用 cd 命令定位到该目录下，按照下面的步骤依次执行。在每个步骤执行完毕之后都可以输入 dmesg 命令，查看内核日志文件。

① 输入 make<回车>，调用 gcc 编译这个驱动程序。编译结束后，用 ls 命令可以看到当前路径下多了一个叫 sbull.ko 的文件，这是编译后得到的设备驱动程序。

② 输入 insmod　sbull.ko<回车>，让操作系统加载这个设备驱动程序。

③ 输入 lsmod<回车>，可以看到目前操作系统中已经加载的模块，其中应该包括我们的 sbull。

④ 输入 ls　/dev<回车>，可以看到在/dev 下自动生成了从 sbulla 到 sbulld 的 4 个设备文件，可以把它们看成 4 个空白盘。

⑤ 输入 mke2fs　/dev/sbulla<回车>，在 sbulla 这个空白盘上建立一个 EXT2 文件系统，相当于 Windows 下的格式化磁盘。请用户不用担心，这里格式化的是一个虚拟盘，不会对我们的硬盘造成任何影响。

⑥ 输入 mount　/dev/sbulla　/mnt<回车>，把这个盘挂接在文件系统的/mnt 目录下。

如果以上步骤都顺利执行完毕，那么恭喜你，现在你的/mnt 目录就是一个容量 512KB 的虚拟磁盘了，可以像操作其他盘一样建立子目录，复制文件等。只是这个盘是虚拟的，在我们的驱动程序卸载后，里面的内容就会全部清除。

⑦ 输入 df<回车>，查看系统中目前挂接的磁盘（更确切的说法应该是，目前挂接的文件系统，但是“磁盘”的说法比较容易理解，下同）。这时应该看到至少有两个磁盘，一个是用户的硬盘，另一个就是刚才挂接的 sbulla 虚拟盘。另外，如果用户的系统中还挂接了软盘、光盘、U 盘等，也应该在这里有所体现。

⑧ 输入 cp　sbull.c　/mnt<回车>，把 sbull.c 文件复制到虚拟盘上，当然这里也可以是其他大小不超过 512KB 的文件（严格地说，还要稍小，因为文件系统元数据本身占据空间）。

⑨ 输入 mkdir　/mnt/dir1<回车>，在虚拟盘上建立一个叫 dir1 的子目录，当然这个子目录的名字可以改成其他用户喜欢的名字。

⑩ 输入 ls　/mnt<回车>，可以看到用户刚才复制的文件及建立的子目录。

⑪ 输入 umount /dev/sbulla<回车>，从系统中卸载这个磁盘。之后再用 ls 命令，已经不能看到上面复制的文件和建立的子目录。

⑫ 输入 rmmod　sbull<回车>，卸载 sbull 驱动程序。卸载后再用 lsmod 命令可以看到，加载模块列表里面已经没有 sbull 这一项，系统恢复到了最初的状态。用户的实验也到此告一段落。

通过上面这个有趣的实验，我们看到了 sbull 这个设备驱动程序的作用。但实际上，sbull 这个程序并不复杂。下面我们打开 sbull.c 文件，一起来分析这个驱动程序的主要工作过程，揭开它的神秘面纱。

在模块初始化函数 sbull_init 中，首先调用 register_blkdev 函数，以对块设备进行注册。实际上，在 2.6 的 Linux 内核里，函数 register_blkdev 完成的功能很少，不一定要调用它。接下来分配若干 sbull_dev 结构，来描述每一个 sbull 设备。然后调用 setup_device 函数，来初始化每一个 sbull 设备。其中要建立请求队列和 gendisk 成员变量，然后调用 add_disk 将此块设备加入系统。之后，该块设备随时都可能接受访问。应该注意的是，Ramdisk 设备并不存在热插拔的问题，因此可以在 sbull_init 函数中建立所有设备的数据结构，最后在 sbull_exit

函数中调用 put_disk 删除这些设备。也就是说，一旦模块加载，Ramdisk 设备就存在了；一旦模块卸载，这些设备自然也需要卸载，同时释放所有资源。但是，对于 U 盘而言，需要作出调整。U 盘可以随时热插拔，当 U 盘未连入系统时，U 盘是不可访问的，因此不能在 sbull_init 函数中进行 add_disk 操作，而应该调整到 USB 设备的 probe 函数中；类似地，put_disk 操作也应该在 U 盘拔掉时调用，即在 USB 设备的 disconnect 函数中，见随后的 10.4.2.2 小节。

图 10-5清楚地说明了数据结构sbull_dev，gendisk和block_device_operations及相关操作之间的关系。图中的虚线说明语句“dev->gd->queue = dev->queue;”和语句“dev->queue = blk_init_queue(sbull_request, &dev->lock);”等。也就是说，dev->gd的请求队列实际上就是dev的请求队列；且后者的请求处理函数为make_request类型的。依据不同的请求模式设置，可以有三种请求处理办法，分别用sbull_full_request，sbull_make_request和sbull_request三个函数表示。这些函数最终都是靠调用函数sbull_transfer来完成实际数据的读/写。对于本实验而言，我们不重点关注效率，因此可以直接使用最简单的RM_SIMPLE模式，这样只需要函数sbull_request和sbull_transfer。而sbull_full_request，sbull_make_request，sbull_xfer_request和sbull_xfer_bio这 4 个函数就不需要了，这是因为在RM_SIMPLE模式下它们根本不会执行。函数sbull_transfer原来是直接读/写内存中的数据，本实验要求直接读/写U盘，因此应该在其中调用函数my_Bulk_transport。至于dev->gd->fops所指向的 5 个函数，它们主要是进行一些控制操作，以实现计数和同步互斥处理，在本实验中不需要多改。

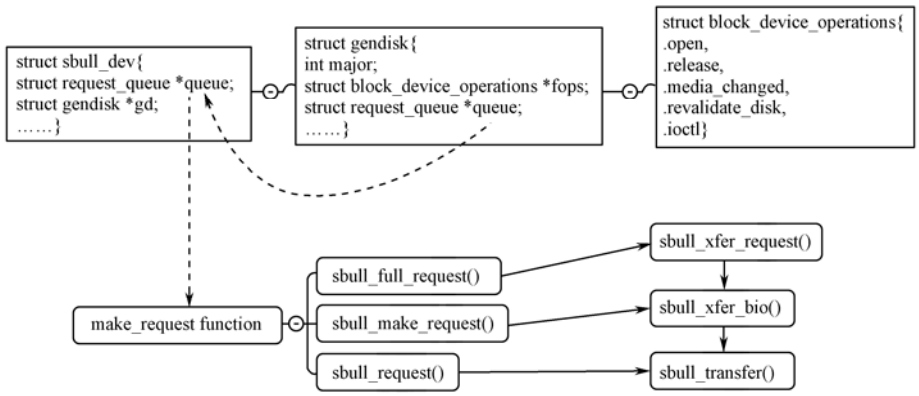

图 10-5　sbull.c 分析图

10.4.2.2　参考驱动程序 2 ——USB字符设备驱动程序usb-skeleton

在 Linux 内核源码目录中，driver/usb/usb_skeleton.c 为我们提供了一个最基础的 USB 驱动程序，我们称为 USB 骨架。通过它我们仅需要修改一些部分，就可以完成一个 USB 设备的驱动程序。

Linux USB 驱动程序需要做的第一件事情就是在 Linux USB 子系统里注册，并提供一些相关信息，如这个驱动程序支持哪种设备，当被支持的设备从系统插入或拔出时，会有哪些动作，所有这些信息都传送到 USB 子系统中。注册通常在驱动程序的初始化函数里完成，需要调用 usb_register 函数。当从系统中卸载驱动程序时，需要注销 USB 子系统，即需要调用 usb_unregister 函数。

当 USB 设备插入时，为了使 Linux 中 PCI，USB 等设备热插拔系统自动装载驱动程序，

需要创建一个 MODULE_DEVICE_TABLE。USB_DEVICE 宏利用厂商 ID 和产品 ID 为我们提供一个设备的唯一标识。当系统插入一个与 ID 匹配的 USB 设备到 USB 总线时，驱动程序中的 probe 函数也就会被调用。usb_device 结构指针和接口 ID 都会被传递到函数中。驱动程序需要确认是否接受插入的设备，如果不接受，或者在初始化的过程中发生任何错误，probe 函数返回一个错误码；否则返回一个 NULL 值。对于一款不明型号的 USB 设备，可先将其插入 USB 口，然后执行命令 dmesg，即可通过内核日志查看连接位置，得到下列信息：

usb 1-1: new full speed USB device using uhci_hcd and address 4

usb 1-1: configuration #1 chosen from 1 choice

然后，对于此设备，就可以在目录/sys/bus/usb/devices/1-1/下找到当前系统探测到的 USB 总线上的设备信息，包括 idVendor 和 idProduct。

如果设备从 USB 总线拔掉，设备指针会调用 disconnect 函数。驱动程序需要清除那些已分配的所有私有数据，关闭 USB 设备，并且注销自己。值得注意的是，USB 设备可以在任何时间从系统中取走，即使程序目前正在访问它。一个优秀的 USB 驱动程序要能够很好地解决此问题，切断任何当前的读/写，同时通知用户空间程序 USB 设备已经被取走。

对于本实验，probe 和 disconnect 两个函数是需要在合并时仔细考虑的。其他一些函数，如 skel_read，skel_write，skel_open 和 skel_release，是针对字符设备的，可以不予考虑。在这些函数中，当需要对 USB 设备进行读/写的时候，调用了 usb_bulk_msg，usb_alloc_urb，usb_submit_urb 和 usb_free_urb 等函数。在本实验中，因为我们提供了粒度更大的 U 盘访问函数 my_Bulk_transport，所以大家不必直接调用这些函数。

10.4.3 U盘驱动的帮助函数

为了帮助大家编写 U 盘驱动程序，降低复杂性，我们以 Linux 下的通用 USB 驱动程序 usb-storage 为基础，整理出了一个 U 盘访问函数 my_Bulk_transport。此函数主要借鉴的是如下三个文件：

① drivers/usb/storage/transport.c

② drivers/usb/storage/transport.h

③ drivers/scsi/scsi.h

该函数的源程序放在文件 mytransport.c 和 mytransport.h 中，读者可在网址 http://nlp.nudt.edu.cn/yjwen/os/mytransport.rar 下载或联系邮箱 yjwen@nudt.edu.cn 来获取。

10.4.3.1 函数原型及其使用

该函数的原型如下所示：

```
int my_Bulk_transport(struct usb_device *udev,
                      struct usb_interface    *interface,
                      struct cmnd_struct *cs,
                      unsigned char *bulk_buffer, int bulk_size,
                      __u8 bulk_in_endpointAddr, __u8 bulk_out_endpointAddr);
```

其功能是向 USB 设备 udev 的接口 interface 发送一个命令，进行对应的数据传递，并获

取反馈的状态信息。各参数解释如下。

① udev 和 interface：U 盘的设备和接口指针，可直接使用 usb_skel 中的对应变量。这里有必要澄清几个基本概念。USB 协议规定，分 4 级来描述 USB 设备：一个 USB 设备（device）可能有很多配置(configuration)；每种配置下又可表现出多个子设备，每个子设备用一个接口（interface）表示、真正对应于一个驱动程序；而子设备与 USB 控制器之间是通过管道（pipe）相连的，而管道的两端称为端点（endpoint）。

② cs：指向一个 cmnd_struct 结构，描述发给 U 盘的命令，这些命令在 UFI 命令规范（USB Mass Storage Class UFI Command Specification）中定义。UFI 命令规范是针对 USB 移动存储而制定的，实际上 UFI 命令格式是基于 SFF-8070i 和 SCSI-2 规范，总共定义了 19 个 12 字节长度的操作命令。具体命令格式及其含义，请查阅 UFI 规范文本。

③ bulk_buffer 和 bulk_size：若上述命令需要有后续的数据传递，则使用缓冲区 bulk_buffer，其最大长度为 bulk_size。

④ bulk_in_endpointAddr 和 bulk_out_endpointAddr：分别用来建立从 USB 控制器到 USB 设备的输入管道和输出管道，进而传递命令（使用输出管道）、数据和状态（使用输入管道）信息。依传输方向的不同，数据的传递既有可能使用输入管道，也有可能使用输出管道。

下面以 U 盘容量获取为例进行解释。首先列出相应的代码片段：

```
/* prepare the command */
memset (dev->command.cmnd, 0, MAX_COMMAND_SIZE);
memset (dev->bulk_buffer,  0,  dev->bulk_size);
dev->command.cmnd[0] = READ_CAPACITY;            /* READ CAPACITY command */
dev->command.cmd_len = 10;
dev->command.sc_data_direction = SCSI_DATA_READ;
/* dev->command.data_payload_size: unused for this command,
   calculated automatically. But for other commands, such as READ_10,
   it may be unommittable */

/* read actually */
result = my_Bulk_transport(dev->udev,  dev->interface,  &dev->command,
                           dev->bulk_buffer,  dev->bulk_size,
                           dev->bulk_in_endpointAddr, dev->bulk_out_endpointAddr);
if (result == MY_TRANSPORT_GOOD) {
    /* now, the capacity parameters are in bulk_buffer.
    * Analize capacity packet:
    * buffer[0] to buffer[3] represent LBN num(n-1).Last Logical Block Address
    * buffer[4] to buffer[7] represent block size in byte.
    */
unsigned char * buffer = dev->bulk_buffer;
my_debug_data (__func__, dev->command.data_payload_actualSize, buffer);

dev->nsectors = ( (buffer[0] << 24) | (buffer[1] << 16) | (buffer[2] << 8) | (buffer[3])) + 1;
                //(__u32*) buffer; block number LBN(n-1)
```

```
dev->hardsect_size = ( (buffer[4] << 24) | (buffer[5] << 16) | (buffer[6] << 8) | (buffer[7]) );
                        //(__u32*)buffer+4;block size.

if (dev->hardsect_size & ((1 << SBULL_SECTOR_BITS)-1)) {
   info("%s : unsupported block size %d.\n", name, dev->hardsect_size);
   dev->nsectors = 0x1fffff;
   dev->hardsect_size = SBULL_SECTOR_SIZE;
   info("%s : block size assumed to be %d bytes, disk size 1GB.\n",
         name, dev->hardsect_size);
} else {
         unsigned int sz = (dev->nsectors/2) * (dev->hardsect_size/256);
         info("USB device %s : ""%u %d-byte hdwr blocks (%u MB)\n",
               name, dev->nsectors, dev->hardsect_size, (sz - sz/625 + 974)/1950);
       }
} else {
      dev->nsectors = 0x1fffff;
      dev->hardsect_size = SBULL_SECTOR_SIZE;
      info("%s : READ CAPACITY failed.", name);
}
```

正如加粗字体所标识的，上述代码片段可分为三大块：准备命令、执行命令和分析结果。命令和结果的具体格式请查阅 UFI 规范，也可参考相关的 Linux 驱动源程序。需要说明的是，对于数据长度固定的命令来说，“dev->command.data_payload_size”可以不设，但对于读/写这类命令来说，则需要设置。另外，my_debug_data，MY_TRANSPORT_GOOD，SCSI_DATA_READ 和 MAX_COMMAND_SIZE 是在 mytransport.h 中定义的，READ_CAPACITY 等命令是在 scsi/scsi.h 中定义的。

上述代码片段输出的内核日志如下，可见其中包括了 31B 的命令、8B 的数据和 13B 的状态信息，结果显示，U 盘容量为 33MB。

```
/root/usb/mytransport.c (my_transfer_length): doDefault 0, data length 8
/root/usb/mytransport.h [command to send]: length = 31, data = << 55 53 42 43 01 00 00 00 08 00 00 00
80 00 0a 25 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 >>
/root/usb/mytransport.c (my_Bulk_transport): Bulk command S 0x43425355 T 0x1 Trg 0 LUN 0 L 8 F
128 CL 10
/root/usb/mytransport.c (my_Bulk_transport): Bulk command transfer result=0, xferred 31/31
/root/usb/mytransport.c (my_Bulk_transport): ----cmnd end

/root/usb/mytransport.c (my_Bulk_transport): xfer 8 bytes data
/root/usb/mytransport.c (my_Bulk_transport): Bulk data transfer result: 0, xferred 8/8
/root/usb/mytransport.c (my_Bulk_transport): ----data end

/root/usb/mytransport.c (my_Bulk_transport): Attempting to get CSW...
/root/usb/mytransport.c (my_Bulk_transport): Bulk status transfer result = 0, xferred 13/13
/root/usb/mytransport.h [status data recerived]: length = 13, data = << 55 53 42 53 01 00 00 00 00 00 00
00 00 >>
/root/usb/mytransport.c (my_Bulk_transport): Bulk status Sig 0x53425355 T 0x1 R 0 Stat 0x0
/root/usb/mytransport.c (my_Bulk_transport): transport indicates command was successful
```

/root/usb/mytransport.h [skel_probe]: length = 8, data = << 00 00 f9 ff 00 00 02 00 >>
/root/usb/main.c: USB device sbulla : 64000 512-byte hdwr blocks (33 MB)

10.4.3.2 工作原理和过程

USB 大容量存储设备通常使用一个带有 USB 接口引擎的微控制器（MPU），处理主机发送的命令以及对存储设备进行操作。USB 组织定义了大容量存储设备的类规范，这个类规范包括 4 个独立的子类规范，即

（1）USB Mass Storage Class Control/Bulk/Interrupt（CBI）Transport

（2）USB Mass Storage Class Bulk-Only Transport

（3）USB Mass Storage Class ATA Command Block

（4）USB Mass Storage Class UFI Command Specification。

前两个子规范定义了数据/命令/状态在 USB 上的传输方法。Bulk-Only 传输规范仅仅使用 Bulk 端点传送数据/命令/状态，CBI 传输规范则使用 Control/Bulk/Interrupt 三种类型的端点进行数据/命令/状态传送。后两个子规范则定义了存储介质的操作命令。

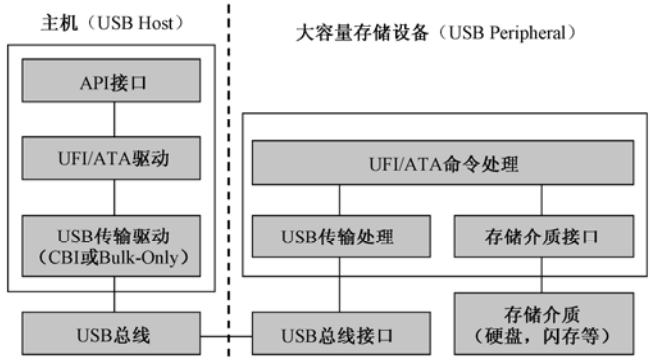

图 10-6　USB 大容量存储设备软件结

图 10-6 是 USB 大容量存储设备软件结构示意图。虚线左边部分是主机的驱动程序结构。顶层是应用程序接口（API），用于提供给用户访问存储设备的方法；中间的 UFI/ATA 驱动层将应用程序的访问转换成 UFI 或 ATA 命令/数据格式，与外部存储设备之间按照子规范 3 或 4 的定义进行命令/状态/数据的交换；底层则是 USB 传输驱动，负责将上层的 UFI/ATA 数据发送到 USB 总线上以及接收从存储设备返回的状态/数据。虚线右边部分是大容量存储设备的固件（Firmware）功能结构。在 USB 总线接口上面是 USB 传输处理层，它与主机之间按照子规范 1 或 2 的定义进行通信，将主机的命令/数据传递到 UFI/ATA 命令处理层并将其状态/数据返回到主机，它还需要检查并处理数据传输过程中的错误。UFI/ATA 命令处理层负责对主机的 UFI/ATA 命令进行处理，并将结果返回给主机。存储介质接口提供与不同存储介质连接的方法，负责将接收到的 UFI /ATA 命令/数据转换成具体的物理信号发送到存储介质，并从存储介质获取状态/数据。

在具体实现上，存储设备可以选择支持两种传输规范（CBI 或 Bulk-Only）或者只支持其中的一种。实际上，Bulk-Only 传输规范是一种更常用的方法。市面上大多数 USB 存储设备都是基于 Bulk-Only 传输规范和 UFI 命令规范设计的。

按照Bulk-Only传输规范，主机与大容量存储设备之间的交互应该遵守 图 10-7 所示的命令（Command）/数据（Data）/状态（Status）协议[10]，即先由主机传输命令给设备，然后进行数据传递（方向依命令而定），接着由设备将命令执行结果状态报给主机，如此反复。该协议同时规定了如果中间出现错误该如何恢复。对于第三步“Status Transport”，还特别定义了一个复杂的交互和恢复流程。

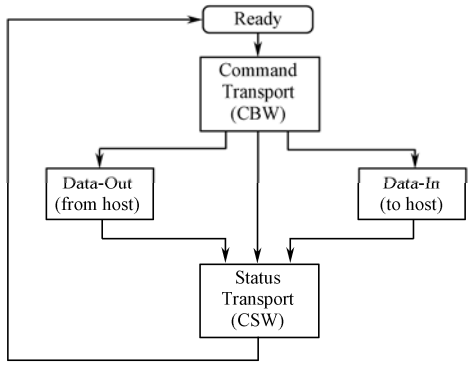

图 10-7　命令/数据/状态协议[10]

事实上，现在来分析 my_Bulk_transport 函数，其关键语句如下所示。容易看出，此函数正是按照上述流程执行的。其中包含许多的错误处理语句，也遵守 Bulk-Only 传输规范。

```
/* set up the command wrapper */
bcb->Signature = cpu_to_le32(MU_BULK_CB_SIGN);
......
bcb->Lun = cs->cmnd[1] >> 5;
bcb->Length = cs->cmd_len;

/* construct the pipe handle */
pipe = usb_sndbulkpipe(udev, bulk_out_endpointAddr);

/* copy the command payload */
memset(bcb->CDB, 0, sizeof(bcb->CDB));
memcpy(bcb->CDB, cs->cmnd, bcb->Length);

/* send it to out endpoint */
result = usb_bulk_msg(udev, pipe, bcb, MU_BULK_CB_WRAP_LEN, &partial,MY_TIMEOUT);
......

/* if the command transfered well, then we go to the data stage */
if (result == 0) {
    /* send/receive data payload, if there is any */
    if (bcb->DataTransferLength) {
        /* calculate the appropriate pipe and buffer information */
        if (cs->sc_data_direction == SCSI_DATA_READ) {
            pipe = usb_rcvbulkpipe(udev, bulk_in_endpointAddr);
            ......
```

```
            }
        } else {
                ......
            }

        /* transfer the data */
        result = usb_bulk_msg(udev, pipe, buf, transfer_amount, &partial, MY_TIMEOUT);
            ......
        }
}

/* construct the pipe handle */
pipe = usb_rcvbulkpipe(udev, bulk_in_endpointAddr);

/* get CSW for device status */
result = usb_bulk_msg(udev, pipe, bcs, MU_BULK_CS_WRAP_LEN, &partial, MY_TIMEOUT);
......
```

10.4.4 实验指南

按照上述方法，将 sbull.c 与 usb-skeleton.c 合并成为一个新文件 main.c，在其中调用 U 盘访问函数 my_Bulk_transport()。然后使用如下 Makefile 文件进行编译：

```
#
# Makefile for the simple-udisk device driver.
#
simple-udisk-objs := main.o mytransport.o
obj-m := simple-udisk.o
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
all:
    $(MAKE) -C $(KERNELDIR) M=$(PWD)
clean:
    rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
```

编译后，会生成一个驱动程序模块 simple-udisk.ko。在加载之前，为避免干扰，最好将系统的通用 USB 驱动程序 usb-storage.ko 备份并删除。如果是 2.6.21 的内核，那么该程序就放在目录/lib/modules/2.6.21/kernel/drivers/usb/storage/下。接着插上 U 盘，执行命令“insmod simple-udisk.ko”以插入模块。如果一切正常，应该可以在内核消息日志中看到下列信息：

```
/root/usb/main.c: USB Skeleton device now attached to sbulla
......
/root/usb/mytransport.c (my_transfer_length): doDefault 0, data length 8
/root/usb/mytransport.h [command to send]: length = 31, data = << 55 53 42 43 01 00 00 00 08 00 00 00
80 00 0a 25 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 >>
/root/usb/mytransport.c (my_Bulk_transport): Bulk command S 0x43425355 T 0x1 Trg 0 LUN 0 L 8 F
```

```
128 CL 10
/root/usb/mytransport.c (my_Bulk_transport): Bulk command transfer result=0, xferred 31/31
/root/usb/mytransport.c (my_Bulk_transport): ----cmnd end
......
/root/usb/mytransport.h [skel_probe]: length = 8, data = << 00 00 f9 ff 00 00 02 00 >>
/root/usb/main.c: USB device sbulla : 64000 512-byte hdwr blocks (33 MB)
......
 sbulla:<7>/root/usb/main.c (sbull_transfer): read sector: sector = 0, nsect = 8

/root/usb/main.c (sbull_transfer): data size requested actually: 4096, blocksize: 512
/root/usb/mytransport.c (my_transfer_length): doDefault 1, data length 4096
/root/usb/mytransport.h [command to send]: length = 31, data = << 55 53 42 43 02 00 00 00 00 10 00 00
80 00 0a 28 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 >>
/root/usb/mytransport.c (my_Bulk_transport): Bulk command S 0x43425355 T 0x2 Trg 0 LUN 0 L
4096 F 128 CL 10
/root/usb/mytransport.c (my_Bulk_transport): Bulk command transfer result=0, xferred 31/31
/root/usb/mytransport.c (my_Bulk_transport): ----cmnd end

/root/usb/mytransport.c (my_Bulk_transport): xfer 4096 bytes data
/root/usb/mytransport.c (my_Bulk_transport): Bulk data transfer result: 0, xferred 4096/4096
/root/usb/mytransport.c (my_Bulk_transport): ----data end

/root/usb/mytransport.c (my_Bulk_transport): Attempting to get CSW...
/root/usb/mytransport.c (my_Bulk_transport): Bulk status transfer result = 0, xferred 13/13
/root/usb/mytransport.h [status data recerived]: length = 13, data = << 55 53 42 53 02 00 00 00 00 00 00
00 00 >>
/root/usb/mytransport.c (my_Bulk_transport): Bulk status Sig 0x53425355 T 0x2 R 0 Stat 0x0
/root/usb/mytransport.c (my_Bulk_transport): transport indicates command was successful
/root/usb/main.c (sbull_transfer): read 4096 bytes successful.
......
unknown partition table
usbcore: registered new interface driver skeleton
```

这些信息依次包括。

（1）USB 设备连接信息。

（2）U 盘容量获取信息，结果显示 U 盘容量为 33MB。

（3）分区表信息，这部分会读出很多扇区，每次读都包括类似前面的命令、数据、状态三部分，最后显示分区表内容不能识别，这是因为实验用的 U 盘没有分区。

（4）USB 设备驱动程序注册信息。

此后，大家就可以对此 U 盘进行通常的 hexdump，mke2fs 和 mount 等操作，在 mount 之后就可以进行各种文件操作了。最后，执行命令“rmmod simple-udisk.ko”卸载模块，同时将反注册 USB 驱动程序，断开设备，在内核日志中会看到下列信息：

```
usbcore: deregistering interface driver skeleton
/root/usb/main.c: SBull #0 now disconnected
```

至此，恭喜你，亲爱的朋友，一个简洁的 U 盘驱动程序就诞生了。

第三部分

Linux环境下的操作及常用命令和函数

Part three

本书的这一部分，归纳了与本书内容相关的非常实用的辅助材料，以方便读者使用时查阅。其中包括 Linux 常用命令和常用函数列表、参数说明和简例，以及内核配置文件生成的相关内容。

附录A　Linux常用命令

A.1　用户终端命令

1．名称：cat

使用权限：所有使用者。

使用方式：cat [option] fileName　。

说明：把文件串连接后传到基本输出（显示器或加 > fileName 到另一个文件）。

option 选项很多，下面列出几个常用选项：

-b 或 --number-nonblank：和–n 相似，只不过对于空白行不编号。

-n 或 –number：由 1 开始对所有输出的行数编号。

-s 或 --squeeze-blank：当遇到有连续两行以上的空白行时，就代换为一行的空白行。

例子：

把 textfile1 的文件内容加上行号后输入 textfile2 这个文件里：

cat -n textfile1 > textfile2

把 textfile1 和 textfile2 的文件内容加上行号（空白行不加）之后将内容附加到 textfile3：

cat -b textfile1 textfile2 >> textfile3

2．名称：cd

使用权限：所有使用者。

使用方式：cd [dirName]。

说明：变换工作目录至 dirName。其中 dirName 表示法可以是绝对路径或相对路径。若目录名称省略，则变换至使用者的 home directory（也就是刚 login 时所在的目录）。另外，“～”也表示 home directory 的意思，“.”表示当前所在的目录，“..”表示当前目录位置的上一层目录。

例子：

跳到 /usr/bin/：

cd /usr/bin

跳到自己的 home directory：

cd ~

跳到当前目录的上上两层：

cd ../..

3．名称：chmod

使用权限：所有使用者。

使用方式：chmod [-cfvR] [--help] [--version] mode file...。

说明：Linux/UNIX 的文件存取权限分为三级：文件拥有者、群组和其他。利用 chmod 可以用来控制文件如何被他人所存取。

mode 为权限设定字串，格式如下：

[ugoa...][[+-=][rwxX]...][,...]，

其中，u 表示该文件的拥有者，g 表示与该文件的拥有者属于同一个群组（group）者，o 表示其他以外的人，a 表示这三者皆是。

+ 表示增加权限，– 表示取消权限，= 表示唯一设定权限。

r 表示可读取，w 表示可写入，x 表示可执行，X 表示只有当该文件是一个子目录或者该文件已经被设定过时为可执行。

-c：若该文件权限确实已经更改，才显示其更改动作。

-f：若该文件权限无法被更改，也不要显示错误信息。

-v：显示权限变更的详细资料。

-R：对当前目录下的所有文件与子目录进行相同的权限变更（即以递归的方式逐个变更）。

--help：显示辅助说明。

--version：显示版本。

例子：

将文件 file1.txt 设为所有人皆可读取：

chmod ugo+r file1.txt

将文件 ex1.py 设定为只有该文件拥有者可以执行：

chmod u+x ex1.py

将当前目录下的所有文件与子目录皆设为任何人可读取：

chmod -R a+r *

此外 chmod 也可以用数字来表示权限，如 chmod 777 file，语法为

chmod abc file

其中，a，b，c 各为一个数字，分别表示拥有者（Owner），组用户（Group）及其他用户（Other）的权限。a，b，c 的值取决于对二进制数（RWX）$_2$ 中二进制位 R，W 和 X 的设置，R/W/X 三位被置位为 1 分别表示允许读/写执行。

例如，要对文件 file 的拥有者设置权限为可读、可写、可执行，而非拥有者则只允许读和执行，不能写，则 a 应该取值为$(117)_2 = 7$，b 应该取值为$(101)_2 = 5$，c 应该取值为$(101)_2 = 5$。所以如下命令便可达到目的：

chmod 755 file

4．名称：cp

使用权限：所有使用者。

使用方式：

cp [options] source dest。

cp [options] source... directory。

说明：将一个文件复制至另一文件或将数个文件复制至另一目录。

options 选项很多，下面列出几个常用选项：

-a：尽可能将文件状态、权限等资料都照原状予以复制。

-r：若 source 中含有目录名，则将目录下的文件也皆依序复制至目的地。

-f：若目的地已经有同名文件存在，则在复制前应先删除同名文件再复制。

例子：

将文件 aaa 复制（已存在），并命名为 bbb：

cp aaa bbb

将所有的 C 语言源代码复制至 Finished 子目录中：

cp *.c Finished

5．名称：cut

使用权限：所有使用者。

用法：cut -cnum1-num2 filename。

说明：显示每行从开头算起的 num1 到 num2 的文字。

例子：

shell>> cat example

test2

this is test1

shell>> cut -c0-6 example　　## print 开头算起的前 6 个字节

test2

this i

6．名称：find。

使用权限：所有使用者。

用法：find [path] [expression]。

使用说明：将文件系统内符合 expression 的文件列出来。可以指定文件的名称、类别、时间、大小、权限等不同信息的组合，只有完全相符的才会被列出来。

find 根据下列规则判断 path 和 expression，在命令列上第一个带“-”参数之前的部分为 path，之后的是 expression。如果 path 是空字串，则使用当前路径；如果 expression 是空字串，则使用-print 为预设 expression。

expression 中可使用的选项有二三十个之多，在此只介绍最常用的部分。

-mount, -xdev：只检查和指定目录在同一个文件系统下的文件，避免列出其他文件系统中的文件。

-amin n：在过去 n 分钟内被读取过。

-anewer file：比文件 file 更晚被读取过的文件。

-atime n：在过去 n 天内读取过的文件。

-cmin n：在过去 n 分钟内被修改过的文件。

-cnewer file：比文件 file 更新的文件。

-ctime n：在过去 n 天内修改过的文件。

-empty：空的文件。

-gid n or -group name：gid 是 n 或是 group，名称是 name 的文件。

-ipath p，-path p：路径名称符合 p 的文件。ipath 会忽略大小写。

-name name，-iname name：文件名称符合 name 的文件。iname 会忽略大小写。

-size n：文件大小是 n 单位，b 代表 512 字节的区块，c 表示字节数，k 表示千字节，w 是两字节。

-type c：文件类型是 c 的文件。

d：目录。

c：字符类型文件。

b：块文件。

f：一般文件。

l：符号连接。

s：socket。

-pid n：process id 是 n 的文件。

expression 中可以使用运算符，下面列出若干：

expl-and exp2：逻辑与，如果 exp1 为假，则不计算 exp2。

!expr：逻辑非，如果 expr 为假，则结果为真。

-not expr：同!expr。

exp1-or exp2：逻辑或，如果 exp1 为真，则不计算 exp2。

exp1,exp2：exp1 和 exp2 均计算，exp1 的值被丢弃，最后的值取决于 exp2 的值。

例子：

将当前目录及其子目录下所有扩展名为 c 的文件列出来：

find . -name "*.c"

将当前目录下子目录中所有一般文件列出：

find . -ftype f

将当前目录及其子目录下所有最近 20 分钟内更新过的文件列出：

find . -ctime -20

7．名称：less

使用权限：所有使用者。

使用方式：less [option] filename。

说明：less 的作用与 more 十分相似，都可以浏览文本文件的内容，所不同的是，less 允许使用者往回卷动，以浏览已经看过的部分，同时因为 less 并未在一开始就读入整个文件，因此在打开大型文件时，会比一般的文本编辑器，如 vi，来得快。

8．名称：more

使用权限：所有使用者。

使用方式：more [-dlfpcsu] [-num] [+/pattern] [+linenum] [fileNames..] 。

说明：类似 cat，不过会一页一页地显示，方便使用者逐页阅读。而其最基本的指令就是按空格键【Space】显示下一页，按【b】键就会往回（back）显示一页，而且还有搜寻字串的功能（与 vi 相似），使用其说明文件，可按【h】键。

-d：提示使用者，在画面下方显示 [Press space to continue, q to quit.]。如果使用者按错键 ，则会显示 [Press h for instructions.]。

-l：取消遇见特殊字节 ^L（送纸字节）时暂停的功能。

-f：计算行数时，以实际的行数而非自动换行后的行数（有些单行字数太长的会被扩展为两行或两行以上）计算。

-p：不以卷动的方式显示每一页，而是先清除屏幕后再显示内容。

-c：跟-p 相似，不同的是，先显示内容，再清除其他旧内容。

-s：当遇到有连续两行以上的空白行时，代换为一行的空白行。

-u：不显示下引号（根据环境变量 TERM 指定的 terminal 而有所不同）。

-num：一次显示的行数。

+/：在每个文件显示前搜寻字串 pattern，然后从该字串之后开始显示。

+linenum：从第 linenum 行开始显示。

fileNames：欲显示内容的文件，如果有多个文件名，它们之间用空格分开。

例子：

more -s testfile

逐页显示 testfile 文件的内容，如有连续两行以上的空白行，则以一行空白行显示。

more +20 testfile

从第 20 行开始显示 testfile 文件的内容。

9．名称：mv

使用权限：所有使用者。

使用方式：

mv [options] source dest。

mv [options] source... directory。

说明：将一个文件重命名，或将数个文件移至另一目录。

options 可以有如下选项：

-i：若目的地已有同名文件，则先询问是否覆盖旧文件。

例子：

将文件 aaa 更名为 bbb：

mv aaa bbb

10．名称：rm

使用权限：所有使用者。

使用方式：rm [options] name...。

说明：删除文件及目录。

options 可以有以下选项：

-i：删除前逐一询问确认。

-f：即使原文件属性设为只读，也直接删除，无须逐一确认。

-r：将目录及其下文件逐一删除。

例子：

删除所有 C 语言程序文件，删除前逐一询问确认：

rm -i *.c

将 Finished 子目录及子目录中所有文件删除：

rm -r Finished

11．名称：rmdir

使用权限：对当前目录有适当权限的所有使用者。

使用方式：rmdir [-p] dirName 。

说明：删除空的目录。

-p：当子目录被删除后使它也成为空目录时，则顺便一并删除。

例子：

将工作目录下名为 AAA 的子目录删除：

rmdir AAA

12．名称：at

使用权限：所有使用者。

使用方式：at -V [-q queue] [-f file] [-mldbv] TIME 。

说明：at 可以让使用者指定在 TIME 这个特定时刻执行某个程序或指令，TIME 的格式是 HH：MM，其中的 HH 为小时，MM 为分钟，甚至可以指定 am，pm，midnight，noon，teatime（就是下午 4 点）等口语词。如果想要指定超过一天内的时间，则可以用 MMDDYY 或者 MM/DD/YY 的格式，其中 MM 是分钟，DD 是第几日，YY 是年份。另外，使用者还可以使用“now+时间间隔”来弹性指定时间，时间间隔可以是 minutes，hours，days，weeks。使用者也可指定 today 或 tomorrow 来表示今天或明天。当指定了时间并按下【Enter】键之后，at 会进入交互模式，并要求输入指令或程序，当输入完后按下【Ctrl+D】键即可完成所有动作，执行的结果将会“寄回”使用者的账号中。

-V：显示出版本编号。

-q：使用指定的队列（queue）来存储 at 的资料，使用者可以同时使用多个 queue，而 queue 的编号为 a，b，c，…，z 以及 A，B，…，Z 共 52 个。

-f：读入预选写好的命令文件 file。使用者不一定要使用交互模式来输入，可以先将所有的指定内容先写入文件后再一次读入。

-m：即使作业执行完成后没有输出结果，也要寄封信给使用者。

-1：列出用户待处理的作业（at-1 等价于命令 atq）。

-d：删除指定的作业（at-d 等价于命令 atrm）。

-v：显示作业将执行的时刻。

例子：

三天后的下午 5 点钟执行/bin/ls：

at 5pm + 3 days /bin/ls

三个星期后的下午 5 点钟执行 /bin/ls：

at 5pm + 2 weeks /bin/ls

明天的 17：20 执行/bin/date：

at 17：20 tomorrow /bin/date

1999 年的最后一天的最后一分钟打印出 the end of the centrury !

at 23：59 12/31/1999 echo the end of the century !

13．名称：passwd

使用权限：所有使用者。

使用方式：passwd [option] [username]。

说明：用来更改使用者的密码。

option 选项很多，下面列出两个：

-d：关闭使用者的密码认证功能，使用者在登录时不必输入密码，只有具备 root 权限的使用者方可使用。

-S：显示指定使用者的密码认证种类，只有具备 root 权限的使用者方可使用[username]选项指定账号名称。

14．名称：who

使用权限：所有使用者。

使用方式：who - [husfV] [user]。

说明：显示系统中有哪些使用者正在上面，显示的信息包含了使用者 ID、使用的终端机、从哪里连上来的、上线时间、滞留时间、CPU 使用量、动作等。

-h：不显示标题列 。

-u：不显示使用者的动作/工作。

-s：使用简短的格式显示 。

-f：不显示使用者的上线位置。

-V：显示程序版本。

15．名称：mail

使用权限：所有使用者。

使用方式：mail [-iInv] [-s subject] [-c cc_addr] [-b bcc_addr] user1 [user 2 ...] 。

说明：mail 不仅只是一个指令，它还是一个电子邮件程序。对于系统管理者来说，mail 很有用，因为管理者可以用 mail 写成 script，定期寄一些备忘录提醒系统的使用者。

i：忽略终端的中断信号。

I：强迫设成互动模式。

v：打印出详细信息，如送信的地点、状态等。

n：不读入 mail.rc 设定文件。

s：subject 为邮件标题。

c：cc_addr 为邮件副本收件人的邮件地址，cc_addr 一栏中的所有用户收到邮件时可以看到彼此的邮件地址。

b：bcc_addr 为密件副本收件人的邮件地址，该栏中的用户看不到其他人的邮件地址。

例子：

将信件送给一个或一个以上的电子邮件地址，由于没有加入其他的选项，使用者必须输入标题和信件的内容等。而 user2 没有主机位置，就会送给邮件服务器的 user2 使用者。

mail user1@email.address user2

将 mail.txt 的内容寄给 user2，同时抄送给 user1：

mail -s 标题 -c user1 user2 < mail.txt

16．名称：kill

使用权限：所有使用者。

使用方式：

kill [-s signal | -p] [-a] pid ...。

kill -l [signal]。

说明：kill 送出一个特定的信号（signal）给进程 id 为 pid 的进程，后者根据该信号做出特定的动作，若没有指定动作，默认送出终止（TERM）信号。

-s signal：其中可用的信号有 HUP（1），KILL（9），TERM（15），分别代表重运行、杀掉、结束。详细的信号可以用 kill -l 查看。

-p：打印出 pid，并不送出信号。

-l signal：列出所有可用的信号名称。

例子：

将 pid 为 323 的进程杀掉：

kill -9 323

将 pid 为 456 的进程重运行：

kill -HUP 456

17．名称：ps

使用权限：所有使用者。

使用方式：ps [options] [--help]。

说明：显示瞬间进程（process）的动态 。

ps 的参数非常多，在此仅列出几个常用的 options 选项：

-A：列出所有的进程。

-w：显示加宽，可以显示较多的信息。

-au：显示较详细的信息。

-aux：显示所有包含其他使用者的进程。

aux 项的输出格式：

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

USER：进程拥有者。

PID：进程的 ID。

%CPU：占用的 CPU 使用率。

%MEM：占用的内存使用率。

VSZ：占用的虚拟内存大小。

RSS：占用的内存大小。

TTY：终端的次设备号（minor device number of tty）。

STAT：该进程的状态。

D：不可中断的静止（如进行 I/O 动作等）。

R：正在执行中。

S：静止状态。

T：暂停执行。

Z：不存在但暂时无法消除。

W：没有足够的内存分页可分配 。

<：高优先序的进程。

N：低优先序的进程。

L：有内存分页分配并锁在内存里。

START：进程开始时间。

TIME：执行时间。

COMMAND：所执行的指令。

18．指令：clear

用途：清屏。

A.2 vi编辑器的用法

在各种操作系统中，编辑器都是必不可少的部件。在 UNIX 和与其相似的 Linux 操作系统中，为方便用户在各种不同的环境中使用，提供了一系列的编辑器软件，包括 ex，edit，ed 和 vi。其中 ex，edit，ed 都是行编辑器，使用不便，现在已经很少有人使用，而 vi 的使用较为普遍。

在系统提示字符（如$，#）下，输入 vi ＜文件名＞，vi 可以自动载入所要编辑的文件或打开一个新文件（如果该文件不存在，或缺少文件名）。进入 vi 后屏幕左方会出现波浪符号，列首有该符号就代表此列目前是空的。

vi 有两种模式：指令模式和输入模式。在指令模式下，输入的按键将作为指令来处理，如输入 a，vi 就认为是在当前位置插入字符。而在输入模式下，vi 把输入的按键作为插入的字符来处理。指令模式切换到输入模式只需输入相应的命令如 a，A，而要从输入模式切换到指令模式，则需在输入模式下键入【Esc】键，如果不知道现在是什么模式，可以多按几次【Esc】键，系统如发出响铃声就表示已处于指令模式下。

在指令模式下，有以下几种编辑命令：

a：从光标所在位置后面开始新增内容，光标后的内容随新增内容向后移动。

A：从光标所在列最后面的地方开始新增内容。

i：从光标所在位置前面开始插入内容，光标后的内容随新增内容向前移动。

I：从光标所在列的第一个非空白字节前面开始插入内容。

o：在光标所在列下新增一列并进入输入模式。

O：在光标所在列上方新增一列并进入输入模式。

在指令模式下键入“:q”、“:q!”、“:wq”或“:x”（注意前面的冒号），就会退出 vi。其中“:wq”和“:x”是存盘退出，而“:q”是直接退出，如果文件已有新的变化，vi 会提示用户保存文件，这时用户可以用“:w”命令保存文件后再退出，也可以用“:wq”或“:x”命令保存后退出。如果不想保存改变后的文件，可以用“:q!”命令，这个命令将不保存文件而直接

退出 vi。

在指令模式下，vi 的基本编辑命令有以下几种：

x：删除光标所在字符。

dd：删除光标所在的列。

r：修改光标所在字节，r 后面接着要修正的字符。

R：进入替换状态，新增文字会覆盖原有文字，直到按【Esc】键回到指令模式为止。

s：删除光标所在字节，并进入输入模式。

S：删除光标所在列，并进入输入模式。

事实上，在 PC 上使用 vi 时，输入和编辑功能都可以在输入模式下完成。例如，要删除某字节，可以直接按【Delete】键，而插入状态与替换状态可以直接用【Insert】键切换，不过如前面所提到的，这些指令几乎是每台终端机都能用，而不仅仅用在 PC 上。在指令模式下移动光标的基本指令是 h，j，k，l，在 PC 上直接用方向键就可以了，当然 PC 键盘也有不足之处。vi 还有一个很好用的指令，用 u 可以恢复被删除的文字，而用 U 指令则可以恢复光标所在列的所有改变。这与某些系统中的【Undo】键功能相同。

这些编辑指令非常有弹性，可以说它们是由指令与范围所构成的。例如，dw 是由删除指令 d 与范围 w 组成，代表删除一个字或单词 d（elete）w（word）。

vi 常见的指令列表如下：

d：删除（delete）。

y：复制（yank）。

p：放置（put）。

c：修改（change）。

v：开始选取。

范围可以是下列几个：

e：光标所在位置到该字的最后一个字母。

w：光标所在位置到下一个字的第一个字母。

b：光标所在位置到上一个字的第一个字母。

$：光标所在位置到该列的最后一个字母。

0：光标所在位置到该列的第一个字母。

)：光标所在位置到下一个句子的第一个字母。

(：光标所在位置到该句子的第一个字母。

}：光标所在位置到该段落的最后一个字母。

{：光标所在位置到该段落的第一个字母。

值得注意的一点是，删除与复制都会将指定范围的内容放到暂存区里，然后用指令 p 贴到其他地方，这是 vi 用来处理区段复制与移动的办法。

某些 vi 版本，如 Linux 所用的 elvis 可以简化这些指令。光标所在位置会反白，然后可以移动光标来设定范围，直接下指令进行编辑即可。

cc 可以修改整列文字；而 yy 则是复制整列文字；指令 D 则可以删除光标到该列结束为止所有的文字。

除了 h，j，k，l 这 4 个基本光标移动指令之外，还可以使用以下命令：

0：移动到光标所在列的最前面。

$：移动到光标所在列的最后面。
【Ctrl+d】：向下半页。
【Ctrl+f】：向下一页。
【Ctrl+u】：向上半页。
【Ctrl+b】：向上一页。
H：移动到视窗的第一列。
M：移动到视窗的中间列。
L：移动到视窗的最后列。
b：移动到下一个字的第一个字母。
w：移动到上一个字的第一个字母。
e：移动到下一个字的最后一个字母。
^：移动到光标所在列的第一个非空白字节。

附录B　Linux常用函数

B.1　进程管理函数

1．pid_t fork()

功能：创建一个新进程，运行与当前进程相同的程序。

参数表：空。

返回值：若失败，返回值为-1；否则，父进程返回子进程的 pid 号，子进程返回 0。

头文件：<sys/types.h>

2．pid_t waitpid(pid_t pid, int* status,int options)

功能：使父进程的执行得以保持，直到子进程退出（即使父进程先处理完，也要等到子进程结束）。子进程退出时，父进程收集子进程的信息，并继续执行直到退出。

参数表：

pid：子进程的 id 号。

status：返回参数，保留子进程结束时的状态信息。

options：进程等待选项，可以为 WCONTINUED，WNOHANG，WNOWAIT 或 WNUTRACED。

返回值：如果成功，返回被等待的子进程的 pid；否则，若指定了 WNOHANG 标志，并且子进程状态无效，则返回零，其余的失败将返回-1。

头文件：<sys/types.h> <sys/wait.h>

3．exec系列函数

完成对其他程序的调用。用户在调用 exec 函数时，进程的当前映像将被替换成新的程序。换言之，如果成功调用了一个 exec 函数，函数的调用不会返回，而新的进程将覆盖原进程所占用的内存空间。通常用子进程调用运行 exec 函数，父进程等待直到子进程结束。

exec 系列函数有 execl，execlp，execle，execv 等。

示例：创建一个进程，由子进程调用命令 ls，父进程等待子进程结束后退出，并以特定方式显示输出。

```
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
```

```
int tprintf(const char *fmt, ...);
void waitchildren(int signum);

int main(void){
	pid_t pid;  //调用进程号
	pid = fork();	//创建进程
	if (pid == 0){	//子进程
		tprintf("Hello from the child process!\n");
		setenv("PS1", "CHILD \\$ ", 1);
		tprintf("I'm calling exec.\n");
		execl("/bin/ls", "/bin/ls", "-l","/etc",NULL);  //调用 ls 命令
		tprintf("You should never see this because the child is already gone.\n");
	}
	else if (pid != -1){	//父进程
		tprintf("Hello from the parent, pid %d.\n", getpid());
		tprintf("The parent has forked process %d.\n", pid);
		tprintf("The parent is waiting for the child to exit.\n");
		waitpid(pid, NULL, 0);	//等待子进程结束
		tprintf("The child has exited.\n");
		tprintf("The parent is exiting.\n");
	}
	else {
		tprintf("There was an error with forking.\n");
	}
	return 0;
}

int tprintf(const char *fmt, ...)
{
	va_list args;
	struct tm *tstruct;
	time_t tsec;
	tsec = time(NULL);
	tstruct = localtime(&tsec);
	printf("%02d:%02d:%02d %5d| ",
		tstruct->tm_hour,
		tstruct->tm_min,
		tstruct->tm_sec,
		getpid());
	va_start(args, fmt);
	return vprintf(fmt, args);
}
```

B.2 文件管理函数

1．int stat(char* path,struct stat sbuf)或int lstat(char* path, struct stat sbuf)

功能：这两个函数用这些信息来填充一个类型为 struct stat 的结构。这两个函数的区别在于 lstat 不对符号链接进行追踪，而只是返回链接本身的信息；而 stat 对符号链接进行追踪，直到链接的最末端。

参数表：

path：需要获得信息的文件路径。

sbuf：返回的文件信息（若调用成功）。

返回值：如果该函数被正确执行，则返回 0；否则，返回–1。

头文件：<sys/stat.h>

2．int open(char* path,int flags,mode_t mode)

功能：打开一个文件，并返回文件的 fd。

参数表：

path：要打开的文件名。

flags：打开文件的方式。可用的值有 O_CREAT，O_RDONLY，O_WRONLY，O_RDWR，O_NONBLOCK，O_APPEND，O_TRUNC，O_EXECL，O_SHLOCK 以及 O_EXLOC 或者合适的组合。

mode：打开文件的权限，请参考附录 A 中的 chmod 命令一节。

返回值：如果成功地打开参数指定的文件，返回该文件的 fd；否则，返回–1。

头文件：<fcntl.h>

3．int read(int fd,void* buf,int count)

功能：从指定文件中读取特定长度的内容至某缓冲区。注意，该文件必须具有读权限。

参数表：

fd：指定文件的 fd。

buf：目标缓冲区。

count：需要读入的数目（以字节计）。

返回值：实际读入的字节数。

头文件：<fcntl.h>

4．int write(int fd,void*buf,int count)

功能：将指定的数据以指定的大小写入指定的文件。同样要求该文件必须具有写权限。

参数表：

d：指定文件的 fd。

uf：源缓冲区。

count：需要写入的数目（以字节计）。

返回值：实际写入的字节数。

头文件：<fcntl.h>

5．int close(int fd)

功能：关闭指定文件。
参数表：
fd：指定文件的 fd。
返回值：如果成功地关闭文件，则返回 0；否则，返回–1。
头文件：<fcntl.h>

6．int chdir(const char *path);

功能：改变工作目录。
参数表：
path：新的当前目录。
返回值：如果成功，则返回 0；否则，返回–1。
头文件：<unistd.h>

7．int dup(int filedes); 和int dup2(int filedes, int filedes2);

功能：复制已有的文件描述符 filedes。
参数表：
filedes：已经打开的文件描述符。
filedes2：被指定的新文件描述符（对于 dup，新文件描述符由系统选定）。
返回值：如果成功，返回新的文件描述符；否则，返回–1。
头文件：<unistd.h>

8．long lseek(int fd,int count,int flags)

功能：将指定文件的读/写指针移动特定的偏移量。
参数表：
fd：目标文件的 fd。
count：偏移量的大小。
flags：文件指针移动选项，可选值有 SEEK_SET，SEEK_CUR，SEEK_END，分别表示文件开始位置、文件指针当前位置以及文件结束位置。
返回值：文件指针的新位置。
头文件：<fcntl.h> <io.h>
示例：打开一个文件，对该文件进行读/写操作，并将文件指针移到指定的位置。

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
```

```
int main(void){
    int fd;      //文件描述符定义
    char *buffer;
    char show[80];
    int len1,len;
    buffer="It is a test!";
    if ((fd=open("test1.c",O_CREAT | O_RDWR))==-1) //文件打开
        printf("cannot open file !\n");
        exit(0);
    }
    len1=write(fd,buffer,strlen(buffer));      //对文件进行写操作
    lseek(fd,3,SEEK_SET);                      //文件指针移动
    len=read(fd,show,80);                      //对文件进行读操作
    show[len+1]='\0';
    printf("file is:%s, len is %d.\n",show,len);
    close(fd);                                 //关闭文件
    return 0;
}
```

B.3 进程间通信

9．int pipe(int fd[0 或 1])

功能：创建一个管道和两个文件描述符。

参数表：

fd：管道文件的描述符数组。其中 fd[0]文件描述符将用于读操作，而 fd[1]文件描述符将用于写操作。

返回值：成功，返回值是 0；失败，将返回–1。

头文件：标准头文件组。

示例：创建管道，子进程向管道写信息，父进程从管道中读取信息。

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdarg.h>
#include <time.h>
#include <string.h>

#define FD_READ 0
#define FD_WRITE 1

void parent(int pipefds[2]);
void child(int pipefds[2]);
int write_buffer(int fd, const void *buf, int count);
```

```
int read_buffer(int fd, void *buf, int count);
int readnlstring(int socket, char *buf, int maxlen);
int readdelimstring(int socket, char *buf, int maxlen, char delim);
int tprintf(const char *fmt, ...);

int main(void){
        int pipefds[2];                                 //定义管道文件描述符
        pipe(pipefds);                                  //管道创建
        if (fork())                                     //进程创建
                parent(pipefds);
        else
                child(pipefds);
        return 0;
}

void parent(int pipefds[2]){                            //父进程从管道中读取信息。
        char buffer[100];
        close(pipefds[FD_WRITE]);                       //关闭管道写
        tprintf("The parent is ready.\n");
        //等待数据，从管道中读取信息，并显示
        while(readnlstring(pipefds[FD_READ], buffer, sizeof(buffer)) >= 0) {
                tprintf("Received message: %s\n", buffer);
        }
        tprintf("No more data; parent exiting.\n");
        close(pipefds[FD_READ]);                        //关闭管道读
}

void child(int pipefds[2]){                             //子进程向管道写信息。
        char buffer[100];
        /* First, close the descriptor that the child doesn't need. */
        close(pipefds[FD_READ]);                        //关闭管道读

        tprintf("The child is ready.\n");
        tprintf("Enter message (Ctrl+D to exit): ");
        while (fgets(buffer, sizeof(buffer), stdin) != NULL){                   //接收从终端输入的数据
                tprintf("Transmitting message: %s\n", buffer);
                write_buffer(pipefds[FD_WRITE], buffer, strlen(buffer));        //数据写入管道
                tprintf("Enter message (Ctrl+D to exit): ");
        }
        tprintf("Client exiting.\n");
        close(pipefds[FD_WRITE]);                               //关闭管道写
}

int write_buffer(int fd, const void *buf, int count){           //写信息到管道
        const void *pts = buf;
        int    status = 0, n;
        if (count < 0) return (-1);
        while (status != count) {
```

```
            n = write(fd, pts+status, count-status);
            if (n < 0) return (n);
            status += n;
        }
        return(status);
}

int read_buffer(int fd, void *buf, int count)
{       //从管道中读取信息
        void *pts = buf;
        int    status = 0, n;
        if (count < 0) return (-1);
        while (status != count) {
            n = read(fd, pts+status, count-status);
            if (n < 1) return n;
            status += n;
        }
        return (status);
}

int readnlstring(int socket, char *buf, int maxlen)
{
        return readdelimstring(socket, buf, maxlen, '\n');
}

int readdelimstring(int socket, char *buf, int maxlen, char delim)
{
        int status;
        int count = 0;
        while (count < maxlen - 1) {
            if ((status = read_buffer(socket, buf+count, 1)) < 1) {
                printf("Error reading: EOF in readdelimstring()\n");
                return -1;
            }
            if (buf[count] == delim) {              /* Found the delimiter */
                buf[count] = 0;
                return 0;
            }
            count++;
        }
        buf[count] = 0;
        return 0;
}

int tprintf(const char *fmt, ...)
{       //特定输出格式
        va_list args;
        struct tm *tstruct;
```

```
        time_t tsec;
        tsec = time(NULL);
        tstruct = localtime(&tsec);
        printf("%02d:%02d:%02d %5d| ", tstruct->tm_hour, tstruct->tm_min, tstruct->tm_sec, getpid());
        va_start(args, fmt);
        return vprintf(fmt, args);
}
```

B.4 多线程库

10．int pthread_create(pthread_t *thread, const pthread_attr_t * attr, void*(*routines)(*void), void* arg)

功能：创建以 routines 为线程体，以 arg 为参数，具有 attr 线程属性的线程。
参数表：
thread：返回参数，新线程的句柄。
attr：新生成线程的属性，如果值为 NULL，则具有默认的线程属性设置。
routine：线程指定运行的函数，该函数必须具有 void*返回值，void*单参数的形式。
arg：该线程运行函数的参数。
返回值：如果成功的创建该线程，函数返回 0；否则，返回一个非 0 的错误码。
头文件：<pthread.h>

11．int pthread_join(pthread_t thread,void**status)

功能：等待一个线程结束，并返回结束时的状态。
参数表：
thread：被等待的线程。
status：返回参数，线程结束时的状态。
返回值：如果线程成功结束，返回 0；否则，返回非零错误码。
头文件：<pthread.h>

12．int pthread_mutex_lock(pthread_mutex_t *mutex)

功能：如果信号量 mutex 未加锁，则为其加锁；否则，该线程阻塞。
参数表：
mutex：需要加锁的信号量。
返回值：如果成功加锁，返回值为零；否则，返回非零错误码。
头文件：<pthread.h>

13．int pthread_mutex_unlock(pthread_t* mutex)

功能：为指定的信号量解锁。

参数表：

mutex：需要加锁的信号量。

返回值：如果成功解锁，返回值为零；否则，返回非零错误码。

头文件：<pthread.h>

示例：创建一个线程，进行运算并显示输出信息；主程序也对相同的变量运算，输出相应信息。

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int myglobal;
pthread_mutex_t mymutex=PTHREAD_MUTEX_INITIALIZER;    //定义静态互斥

void *thread_function(void *arg)
{
    int i,j;
    for( i=0; i<20; i++) {
        pthread_mutex_lock(&mymutex);                    //加锁
        j=myglobal;
        j=j+1;
        printf(".");
        fflush(stdout);
        sleep(1);
        myglobal=j;
        pthread_mutex_unlock(&mymutex);                  //解锁
    }
    return NULL;
}

int main(void)
{
    pthread_t mythread;
    int i;
    if(pthread_create(&mythread, NULL, thread_function, NULL)) {
        printf("error creating thread.");
        abort();
    }
    for(i=0; i<20; i++) {
        pthread_mutex_lock(&mymutex);        //加锁
        myglobal=myglobal+1;
        pthread_mutex_unlock(&mymutex);      //解锁
        printf("o");
        fflush(stdout);
        sleep(1);
```

```
        }
        if(pthread_join(mythread, NULL)) {
                printf("error joining thread.");
                abort();
        }
        printf("\nmyglobal equals %d\n",myglobal);
        exit(0);
}
```

附录C　内核配置文件的生成

刚解压的内核代码树的顶级目录下还没有.config 文件，有很多方法可以生成它。这里首先介绍初步生成.config 文件，然后介绍修改已有的.config 文件，最后介绍内核配置选项。下面所有命令的工作目录假定是在内核代码树的顶级目录。

C.1　配置文件初步生成

这里介绍三种方法。

第一种方法在实验 6 中已经介绍过，该方法也是我们推荐使用的方法。

第二种方法使用如下命令：

$make defconfig

该命令会自动在当前目录下生成.config 文件，该配置文件往往就是内核维护者在其本人机器上的内核编译配置文件，所以不一定适合用户的机器，可能需要修改某些配置选项。

第三种方法使用如下命令：

$ make config

```
  HOSTLD   scripts/kconfig/conf


scripts/kconfig/conf arch/i386/Kconfig
.....
*
* Linux Kernel Configuration
*
*
* Code maturity level options
*
Prompt for development and/or incomplete code/drivers (EXPERIMENTAL) [Y/n/?]
```

使用“make config”命令会针对内核所有的配置选项逐项询问用户的设置。配置选项以[Y/m/n/?]的形式出现，其中的大写字母是默认值，若想选中它直接【回车】键就可以，其他的选项则要输入相应字符再按【回车】键。不是所有的编译选项都有 4 个选项，也可能只有三个。4 个选项的含义如下：

① y——将该功能编译进内核。

② n——相应功能不编译。

③ m——以内核模块的方式编译。

④ ？——输出该选项的说明信息，再次询问用户的设置。

内核有大约 2000 个内核配置选项，逐项设置需要花费大量时间，不建议采用该方法。

C.2　修改内核配置文件

对使用 C.1 节方法生成的配置文件，我们可以对其中一些选项进行调整。有三个命令可供选择：

① make menuconfig 使用基于 ncurses 的界面，如图 C-1所示。

② make gconfig 使用基于 gtk+的图形界面，如图 C-2 所示。

③ make xconfig 使用基于 qt 的图形界面。

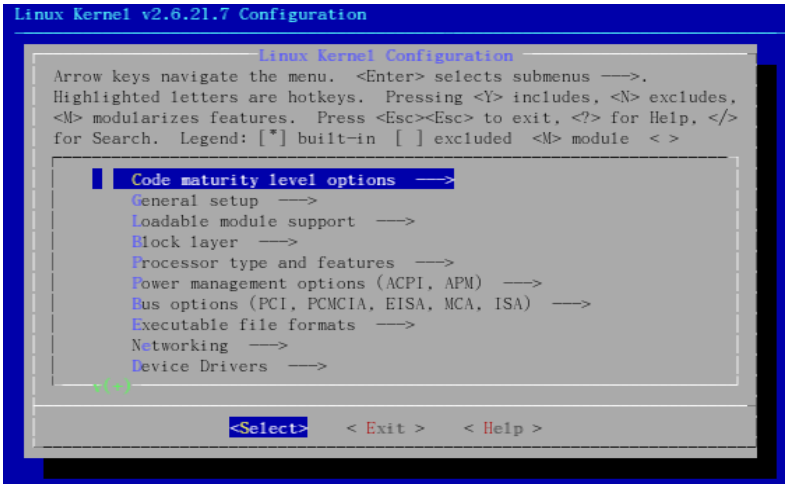

图 C-1　make menuconfig 界面

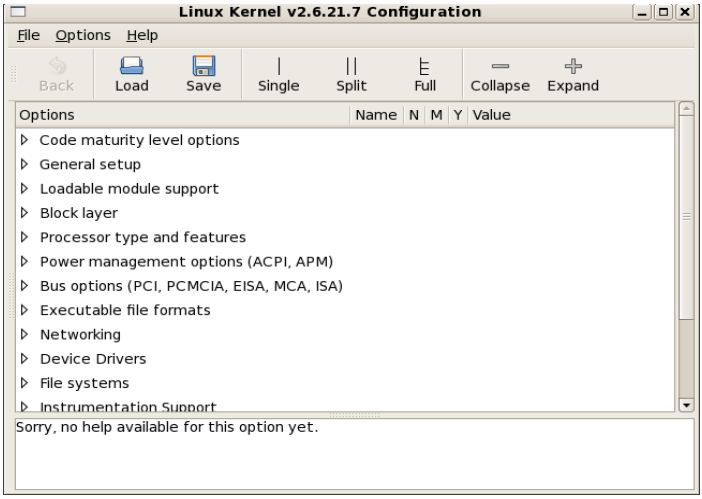

图 C-2　make gconfig 界面

对各发行版本来说，make menuconfig，make gconfig 和 make xconfig 能否使用要看相应的软件包是否已经安装。这三个命令会读取已存在的.config 文件配置，并以图形化方式将值显示出来，用户可以进一步调整。值得一提的是，用户也可以在.config 文件不存在的情况下，

直接使用这几个命令生成配置文件，但是我们不推荐这样做。命令 make menuconfig，make gconfig 和 make xconfig 按照层次结构组织各个内核编译选项，例如 Processor family 就在 Processor type and features 之下。

在 menuconfig 中，操作是通过键盘实现的。屏幕的上方提供了足够的操作帮助信息，如上下键可以选择各节（section），按【回车】键可以进入子菜单，如此下去可以进入底层菜单。光标停在具体的编译选项后，可按【y】、【m】或【n】键进行设置，也可以按【空格】键在这几者之间切换。要跳出某小节，按【Esc】键或者按右方向键【→】选中【Exit】后按【回车】键退出。如果用户已经在主菜单中，系统将退出配置并询问是否保存配置文件。

而 gconfig 和 xconfig 两者界面有类似之处，都是通过鼠标操作来设置编译选项值，相比 menuconfig 操作更简单，再此就不多介绍了，要提醒的是，退出界面时要保存配置文件。

C.3 内核编译选项介绍

内核配置被分成多节，每节包含一个特定的主题，该主题下又包含多个子项。这里只简单介绍如下各大主题。

（1）Code maturity level options 代码成熟等级，对本书来讲选择 N 即可。

（2）General setup 常规设置。这部分内容较多，一般使用默认设置就可以了。

（3）Loadable module support 对模块的支持。其中子选项 Enable loadable module support 一般是要选的。

（4）Block layer Block I/O 层的特性及 I/O 调度算法的选择。

（5）Processor type and features 选择支持的 CPU 类型及其特性。

（6）Power management options 电源管理选项。当前 PC 一般支持 ACPI 标准，一般不选择 APM 标准。

（7）Bus options 总线选项。对 PC 来说 PCI 是必选的。

（8）Executable file formats 可执行文件格式，其中 ELF 格式是必选的。

（9）Networking 网络。包括各种网络协议，当然 TCP／IP 是必选的。

（10）Device Drivers 设备驱动程序。本选项里面包含了各式各样的外部设备，诸如块设备、字符设备、网卡和 USB 设备等，要根据机器具体情况选择。

（11）File systems 文件系统。对 PC 用户来说，应该选 EXT3 和一些伪文件文件系统。

（12）Instrumentation Support 分析支持。用于内核性能分析和探测。

（13）Kernel hacking 内核开发选项。主要提供内核开发的各种调试支持。

（14）Security options 安全选项。选择默认值即可。

（15）Cryptographic options 加密选项。包含了多种加密算法。

（16）Library routines 库函数，供第三方内核模块使用。按默认处理即可。

参 考 文 献

[1] A.S. Tanenbaum. Modern Operating System, 2nd edition. Prentice Hall, 2001

[2] Jonathan Corbert. Linux Device Drivers, 3rd edition. O'Reilly, 2002

[3] Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, 3rd edition. O'Reilly, 2005

[4] W.Richard Stevens 著，杨继张译. UNIX 网络编程（第二版）第 2 卷：进程间通信. 北京：北京科海电子出版社，2000

[5] 毛德操，胡希明. LINUX 内核源代码情景分析. 杭州：浙江大学出版社, 2001

[6] Rémy Card. Design and Implementation of the Second Extended Filesystem. Proceedings of the First Dutch International Symposium on Linux, 1994

[7] Bash 参考手册. http://www.gnu.org/manual/bash/index.html

[8] Peter Jay Salzman. The Linux Kernel Module Programming Guide. http://www.faqs.org/docs/kernel/

[9] 罗宇，褚瑞等. 操作系统课程设计. 北京：机械工业出版社，2005

[10] USB Implementers Forum. Mass Storage device class specification. http://www.usb.org

[11] 罗宇，邹鹏等. 操作系统（第 2 版）. 北京：电子工业出版社，2007